1996

Simulator for the Performance Analysis of CPM Schemes in an Indoor Wireless Environment

Ronald Chua
Edith Cowan University

Edith Cowan University

Copyright Warning

Simulator for the Performance Analysis of CPM Schemes in an Indoor Wireless Environment

A Thesis Submitted in Partial Fulfilment of the Requirements for the Award of Bachelor of Engineering (Electronic Systems) with Honours

Ronald Chua
0948089

Principal Supervisor: Dr Tadeusz Wysocki

June 1996

Faculty of Science, Technology, and Engineering
Department of Computer and Communication Engineering
Edith Cowan University
Western Australia

USE OF THESIS

The Use of Thesis statement is not included in this version of the thesis.

To Jenny

Abstract

A software simulator for characterising Continuous Phase Modulation (CPM) schemes in an indoor multipath environment has been developed using SIMULINK and MATLAB. The simulator is capable of simulating a wide range of CPM schemes to determine bandwidth efficiency and robustness to additive white Gaussian noise (AWGN) and Rician fading. Initial trials of the simulator indicate that the simulator is functioning correctly. Eventually, the simulator will be used to determine the most suitable modulation scheme for the development of an actual indoor wireless system.

I certify that this thesis does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any institution of higher education; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person except where due reference is made in the text.

Signature .....

Date 18/7/96

Acknowledgments

I wish to express my sincere appreciation to Dr Tadeusz Wysocki for his guidance and supervision throughout this project. Although it is with a sense of relief that I have completed this demanding project, I am also very grateful for having undertaken the challenge. As a result of the project, I have acquired a greater understanding of advanced digital modulation systems. I thank Dr Wysocki for exposing me to an interesting project and for providing the necessary support to complete it.

My appreciation also extends to Dr Stephen Fischer of Curtin University of Technology for reviewing this thesis on such short notice and for providing valuable feedback to improve it.

I would like to acknowledge the support of the Australian Telecommunications Research Institute and the Cooperative Research Centre for Broadband Telecommunications and Networking for providing the computer and software facilities to develop the simulators. The work experience and the friendships I developed there has been invaluable.

The completion of this thesis closes a long chapter to obtain a formal qualification. As an immigrant into a foreign country, the struggle has been hard. I graciously thank my parents for forsaking all material interests to immigrate to Australia for the benefit of their children's education. I also thank Australia for the opportunity of a better life for which I am eternally grateful.

Most of all, I thank my fiance Jenny for her steadfast love and tremendous support. She has always believed in me and for that, I dedicate this thesis to her.

Table of Contents

Acronyms

AWGN	Additive White Gaussian Noise
BER	Bit Error Rate
CM	Correlation Metric
CPM	Continuous Phase Modulation
CPFSK	Continuous Phase Frequency Shift Keying
CDMA	Code Division Multiple Access
FSK	Frequency Shift Keying
PSK	Phase Shift Keying
MAP	Maximum A Posteriori
MLSE	Maximum Likelihood Sequence
MLWP	Most Likely Windowed Path
SWP	Surviving Windowed Path
ESWP	Extended Surviving Windowed Path
SNR	Signal-to-Noise Ratio

1 Introduction

1.1 Motivation and Contributions of the Thesis

Indoor wireless communication systems, particularly microwave LANs, have attracted much research interest in the last few years due to the growth of computer based applications in almost all work environments. Of a primary interest, is the benefit of increased mobility offered by microwave LANs which can facilitate a more flexible work environment. However, associated with any indoor microwave implementation, there is the major problem of multipath effects [16]. Multipath effects can cause serious distortions to the received signal. To combat multipath effects, Code Division Multiple Access and Frequency Hopping techniques are employed in the higher levels [13] [14]. However, it is also important to select a robust and efficient modulation scheme to reinforce the effectiveness of the overall physical system.

Continuous Phase Modulation (CPM) is a class of digital modulation schemes in which the phase signal is constrained to be continuous [8]. CPM schemes are highly attractive because they offer better spectral utilisation compared with other modulation schemes. Another benefit of CPM is the built-in memory resulting from the constraint imposed on the phase signal [8]. This memory can be used to improve the performance of the modulation scheme in the presence of noise, interference and distortions [1].

The performance of more popular modulation schemes such as M-ary Frequency Shift Keying (M-ary FSK) and M-ary Phase Shift Keying (M-ary PSK) in both additive white Gaussian noise (AWGN) and Rician channels are well characterised and widely available. However, the performance of more efficient CPM schemes are less publicised. There have been software simulators built to obtain the performance comparisons of CPM schemes such as the one described in [2] and [6]. However, the simulators used there and in most literatures are not flexible enough to cater for CPM schemes in general. Typically, the performance analysis of CPM schemes in published literature are merely

confined to Continuous Phase Frequency Shift Keying (CPFSK) and usually over AWGN channel which do not account for multipath effects and other real channel problems. Also, limiter-discriminator detection techniques are commonly utilised as in [2] instead of the more superior Maximum Likelihood Sequence (MLSE) detector.

The unavailability of a comprehensive performance characterisation of CPM schemes, particularly for the indoor wireless environment has generated the motivation behind the project reported in this thesis. The aim of this project was:

To develop a flexible software simulator for analysing the performance of generic, non multi-h CPM schemes utilising MLSE reception. The model should account for the effects of additive white Gaussian noise, reception filtering and multipath fading in an indoor wireless channel.

This thesis presents an original work in the design and implementation of a simulation testbed for the performance analysis of CPM schemes over a realistic channel in the presence of AWGN and multipath effects. The simulator developed is capable of simulating generic M-ary, pulse shaped CPM schemes in both full and partial response variants. It uses the more superior MLSE detector which employs the Viterbi algorithm [1]. The flexibility of the simulator allows system parameters such as signalling levels, modulation index and frequency pulse functions to be conveniently adjusted during the course of analysis. An automatic Monte Carlo simulation regulator was also devised which can coordinate the simulator to perform bit error performance analysis for the entire range of signal-to-noise ratios unattended. As part of the literature survey conducted for this project, no published material utilising such a comprehensive simulator was found.

The most significant outcome of this project is that a comprehensive evaluation of CPM schemes in an indoor multipath environment can now be performed in order to determine the most suitable scheme for an indoor wireless communication system. The simulator developed in this project will assist in formulating the design criteria for the actual development of such a system.

The simulator was implemented in SIMULINK and MATLAB. By convention, MATLAB function blocks in SIMULINK are commonly used to implement simple functions or otherwise, functions defined by state equation techniques. It is difficult, if not impossible to define complicated processing using only state equations. Thus, MATLAB functions using procedural language techniques were used to develop relatively large and complicated processing blocks in the simulator. Although MATLAB functions utilised in this manner in SIMULINK is an unconventional approach, this method effectively combines the power of block diagram modelling in SIMULINK with the power of procedural language processing in MATLAB. To achieve this amalgamation, a technique of using global variables in MATLAB was developed and used in this project.

1.2 Outline of the Thesis

The layout of this thesis is as follows:

- Chapter 2 provides the theoretical foundation required to understand continuous phase, narrowband digital modulation schemes and then explains CPM transmission and reception techniques together with the expected performance of CPM schemes as functions of varying system parameters. It also presents some common performance metrics and proposes an effective metric for the overall performance analysis of digital modulation schemes.

- Chapter 3 examines the channel environment of a wireless LAN and its effects on the received signal. Subsequently, the composite channel models which are suitable for simulation are presented. In this chapter, the development of a Rician channel model is examined in detail.

- Chapter 4 introduces the simulation software, SIMULINK and its specific benefits. It then discusses the implementation of the simulator and its specialised versions. Every

major component of the simulator is closely examined to provide an accurate representation of the simulator's behaviour.

- Chapter 5 presents some preliminary results from the trials of the simulator. So far, the results obtained indicate that the simulator is performing correctly.

- Finally, Chapter 6 concludes the thesis by summarising the major outcomes of the project and suggests worthwhile avenues for further research work which have been opened by the development of this simulator.

2 Narrowband Digital Modulation

In this chapter, the theoretical background for CPM schemes and its special forms, namely CPFSK and MSK are presented. The generation of CPM schemes, its detection and its performance characteristics are discussed.

CPM schemes are a class of nonlinear digital modulation schemes in which the phase of the signal is constrained to be continuous. This constraint results in a phase or frequency modulator that has memory. CPM schemes are also categorised as narrowband because their signals usually satisfy the condition that their bandwidth is much smaller than the carrier frequency [8].

Some common representations [5] of angle-modulated bandpass signals are provided below. These definitions will be used throughout this thesis. An angle-modulated bandpass signal may be represented by

$$s(t) = A\cos(2\pi f_c t + \phi(t)) \tag{2-1}$$

where A is the constant amplitude of the modulated signal s(t), f_c is the carrier frequency, and $\phi(t)$ is the information bearing phase waveform. Expression (2-1) may be also expressed in its hyperbolic form as

$$s(t) = A\mathrm{Re}\{e^{j(2\pi f_c t + \phi(t))}\} = A\mathrm{Re}\{e^{j2\pi f_c t} e^{j\phi(t)}\} \tag{2-2}$$

where the modulated signal is proportional to the real part of a complex exponential. Another popular form is to represent the signal in its quadrature and in-phase components, defined as

$$\begin{aligned} s(t) &= A\cos(\phi(t))\cos(2\pi f_c t) - A\sin(\phi(t))\sin(2\pi f_c t) \\ &= I(t)\cos(2\pi f_c t) - Q(t)\sin(2\pi f_c t) \end{aligned} \tag{2-3}$$

where $I(t) = A\cos(\phi(t))$ and $Q(t) = A\sin(\phi(t))$ are called the *in phase* and *quadrature* components respectively. The in phase and quadrature components are known as the *baseband components* of the modulated signal since they do not account for the carrier while the terms $\cos(2\pi f_c t)$, $\sin(2\pi f_c t)$ are known as the *carrier components* [5].

2.1 Continuous-Phase Frequency Shift Keying (CPFSK)

[8]

As well as being a special form of CPM, Continuous-Phase Frequency Shift Keying (CPFSK) is also a derivative of Frequency Shift Keying (FSK). Initially in this section, a brief description of FSK is provided, followed by a description of the constraint which is necessary to generate CPFSK from FSK.

To generate a conventional FSK signal, the carrier is shifted by an amount $f_n = \frac{1}{2}\Delta f I_n$, where $I_n = \pm 1, \pm 3, \ldots, \pm(M-1)$ represents the digital information being transmitted. To facilitate the switching from one frequency to another, $M = 2^k$ separate oscillators are used where k is the number of bits per symbol. These oscillators are tuned to the desired frequencies and are selected according to the particular k-bit symbol that is to be transmitted in a signal interval of duration T = k/R seconds where T is also known as the symbol period and R is the bit rate. The adverse effect of this method is that it requires a large frequency band for the transmission of the signal due to the relatively large spectral side lobes outside the main spectral band of the signal. This is caused by the discontinuity in the phase signal due to the abrupt switching from one oscillator output to another during successive signalling intervals. By modulating a single carrier whose frequency is changed continuously we can avert the fore-mentioned problem. This results in a frequency-modulated signal that is phase-continuous, hence, the term continuous-phase FSK. Due to the continuity constraint imposed on the carrier, a CPFSK signal has memory.

A CPFSK signal can be represented by first considering a Pulse Amplitude Modulation (PAM) signal

$$d(t) = \sum_n I_n g(t - nT) \qquad (2.1\text{-}1)$$

where $\{I_n\}$ is derived from the mapping of k-bit blocks of binary digits from the information bearing sequence levels $\{a_n\}$ into the amplitude levels $\pm 1 \pm 3, \ldots, \pm(M-1)$. The function g(t) is a rectangular pulse of duration T seconds and amplitude 1/2T. The

PAM signal d(t) is utilised to frequency-modulate the carrier. The equivalent low pass waveform is expressed as

$$v(t) = \sqrt{\frac{2E}{T}} \exp\left\{ j \left[4\pi T f_d \int_{-\infty}^{t} d(\tau) d\tau + \phi_0 \right] \right\} \tag{2.1-2}$$

where E is the energy per symbol, f_d the peak frequency deviation and ϕ_0 is the initial phase of the carrier. The corresponding carrier-modulated signal is expressed as

$$s(t) = \sqrt{\frac{2E}{T}} \cos\left[2\pi f_c t + \phi(t; I) + \phi_0 \right] \tag{2.1-3}$$

where $\phi(t;I)$ is the time-varying phase of the carrier, defined as:

$$\begin{aligned} \phi(t; I) &= 4\pi T f_d \int_{-\infty}^{t} d(\tau) d\tau \\ &= 4\pi T f_d \int_{-\infty}^{t} \left(\sum_n I_n g(\tau - nT) \right) d\tau \end{aligned} \tag{2.1-4}$$

Although d(t) is a discontinuous function of time, its integral is continuous. Hence, the phase $\phi(t;I)$ is also a continuous function of time. By integrating (2.1-4), the phase of the carrier for the interval $nT \le t \le (n+1)T$ is obtained as

$$\begin{aligned} \phi(t; I) &= 2\pi f_d T \sum_{k=-\infty}^{n-1} I_k + 2\pi f_d (t - nT) I_n \\ &= \theta_n + 2\pi h I_n q(t - nT) \end{aligned} \tag{2.1-5}$$

where h, θ_n, and q(t) are defined as:

$$h = 2 f_d T \tag{2.1-6}$$

$$\theta_n = \pi h \sum_{k=-\infty}^{n-1} I_k \tag{2.1-7}$$

$$q(t) = \begin{cases} 0 & (t < 0) \\ t/2T & (0 \le t \le T) \\ 1/2 & (t > T) \end{cases} \tag{2.1-8}$$

The parameter h is called the modulation index. Note that θ_n represents the accumulation (memory) of all symbols up to the time (n-1)T.

2.2 Continuous-Phase Modulation (CPM)

[8]

From equation (2.1-5), we recognise that CPFSK is a special case of a general class of continuous-phase modulated (CPM) signals where the carrier phase is

$$\phi(t;\mathbf{I}) = 2\pi \sum_{k=-\infty}^{n} I_k h_k q(t-kT), \qquad nT \le t \le (n+1)T \tag{2.2-1}$$

with $\{I_k\}$ being the sequence of M-ary information bearing symbols selected from the alphabet set $\{\pm 1 \pm 3,\ldots,\pm(M-1)\}$, $\{h_k\}$ being a sequence of modulation indices and q(t) being some normalised waveform shape.

The modulation index may be held constant for all symbols such that $h_k = h$ for all k. When the modulation index varies from one symbol to another in a cyclic manner through a set of indices, the CPM signal is called *multi-h*.

In general, the signal waveform q(t) may be represented as the integral of some pulse g(t) called the *frequency pulse*, such that:

$$q(t) = \int_0^t g(\tau)d\tau \tag{2.2-2}$$

If $g(t) = 0$ for $t > T$, the CPM signal is called full response CPM. Otherwise, if $g(t) \neq 0$ for $t > T$, the modulated signal is called partial response CPM.

Figure 2.2-1 below illustrates several pulse shapes g(t) and the corresponding q(t) waveforms. It is evident that an infinite variety of CPM signals can be generated by selecting different pulse shapes g(t) and by varying the modulation index h and the alphabet size M.

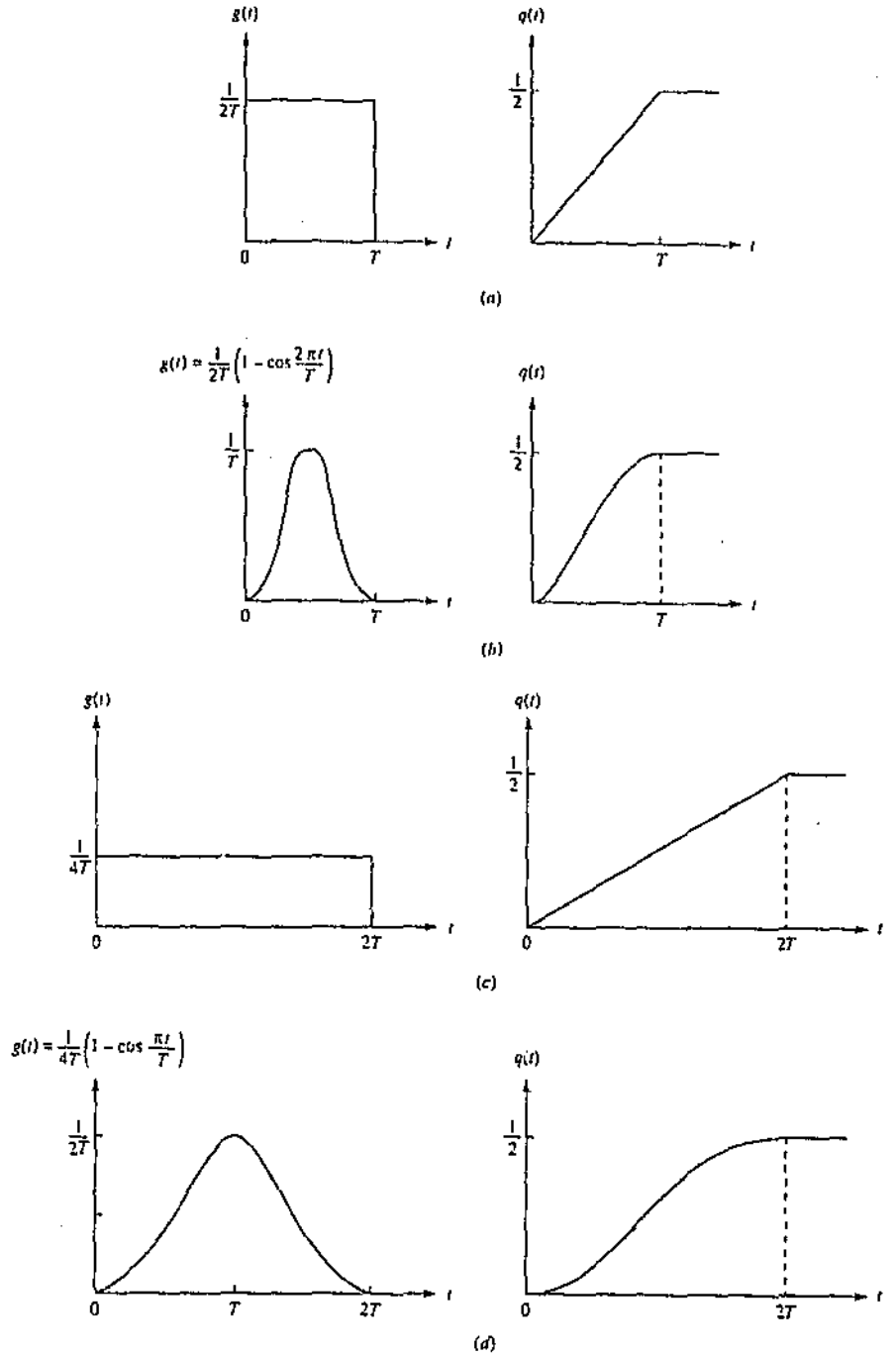

Figure 2.2-1: Pulse shapes for full response CPM (a,b) and partial response CPM (c,d) [8].

Table 2.2-1 presents a further variety of important pulse shaped schemes commonly found in the literature. The prefix L, where it occurs, denotes the length of the pulse g(t) in terms of the symbol interval T. An explanation of the abbreviations used in the table is listed below [12]:

LRC	raised cosine with pulse length L
LSRC	spectrally raised cosine with pulse length L
LREC	rectangular frequency pulse with pulse length L
GMSK	Gaussian-shaped MSK

LRC	$g(t) = \begin{cases} \frac{1}{2LT}\left[1 - \cos\frac{2\pi t}{LT}\right] & ; 0 \le t \le LT \\ 0 & ; otherwise \end{cases}$ L is the pulse length, e.g., 3RC has $L=3$.
LSRC	$g(t) = \frac{1}{LT} \cdot \frac{\sin\left(\frac{2\pi t}{LT}\right)\cos\left(\beta \cdot \frac{2\pi t}{LT}\right)}{\frac{2\pi t}{LT}\left[1-\left(\frac{4\beta}{LT}\cdot t\right)^2\right]} ; 0 \le \beta \le 1$
GMSK	$g(t) = \frac{1}{2T}\left[Q\left(2\pi B_b \frac{t-\frac{T}{2}}{\sqrt{\ell n 2}}\right) - Q\left(2\pi B_b \frac{t+\frac{T}{2}}{\sqrt{\ell n 2}}\right)\right]$ $; 0 \le B_b T < \infty$ $Q(t) = \int_t^\infty \frac{1}{\sqrt{2\pi}} e^{-\tau^2/2}\, d\tau$
LREC	$g(t) = \begin{cases} \frac{1}{2LT} & ; 0 \le t \le LT \\ 0 & ; otherwise \end{cases}$ $L=1$ yields 1REC which is most often referred to as CPFSK.

Table 2.2-1: Definition of some pulse shapes g(t) commonly found in literature. The parameters B_b in GMSK and β in LSRC are arbitrary and are set to obtain the desired distance or spectral properties [12].

A sketch of the set of phase trajectories $\phi(t;\mathbf{I})$ generated by all possible values of the information sequence $\{I_n\}$ can be observed in Figure 2.2-2 for quaternary CPFSK with the quaternary symbols $I_n = \pm 1, \pm 3$. In this Figure, the set of phase trajectories beginning at time t = 0 is shown. This phase diagram is called a *phase tree*. For comparison, Figure 2.2-3 shows the phase tree for a quaternary CPM scheme utilising the raised cosine pulse shape of length L = 2. Figure 2.2-4 shows the phase trajectories for binary CPFSK and binary partial response CPM based on the raised cosine pulse of length 3T. This example shows the information bearing phase functions of the two modulation schemes for the data sequence +1, -1, -1, -1, +1, +1, -1, +1.

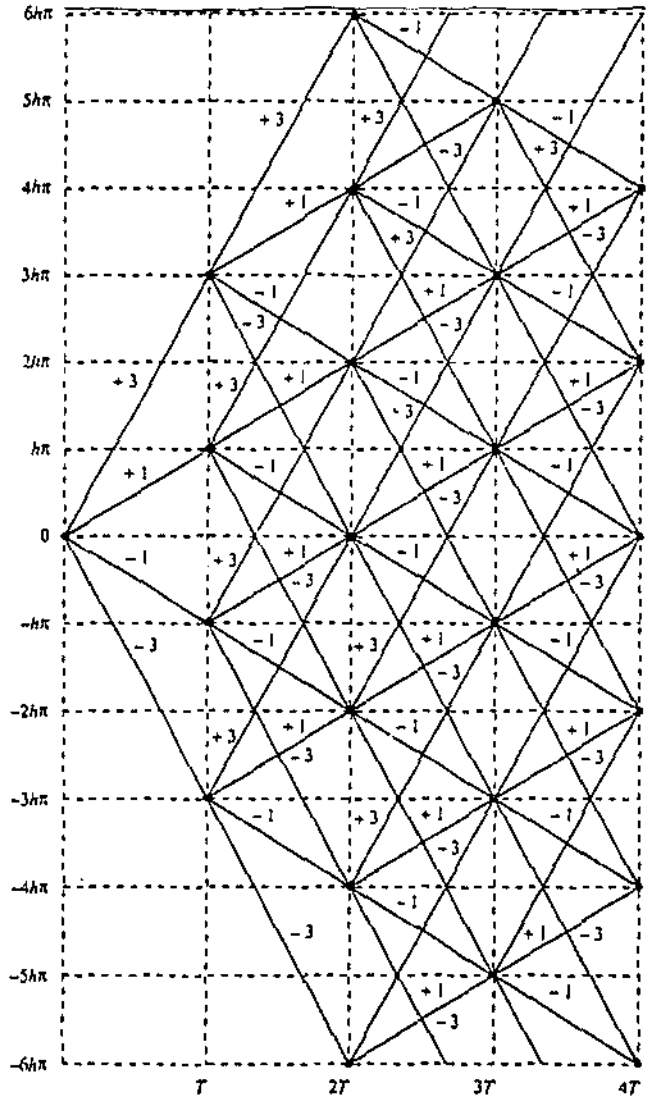

Figure 2.2-2: Phase trajectory for quaternary CPFSK [8].

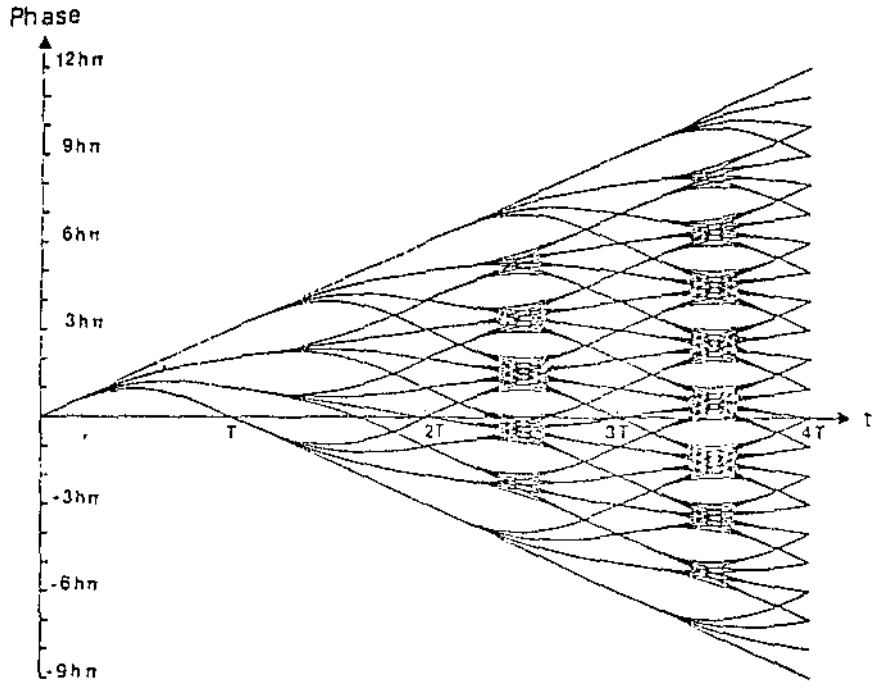

Figure 2.2-3: The phase tree for quaternary scheme 2RC. The state description in the tree will be explained later [1].

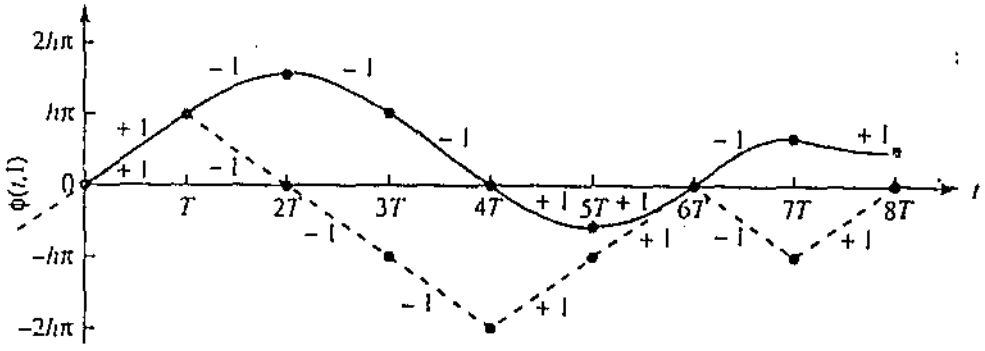

Figure 2.2-4: Phase trajectories for binary CPFSK (dashed line) and binary partial response CPM based on raised cosine pulse of length 3T (solid line) [8].

Although the phase trees shown in figures 2.2-2 and 2.2-3 grow with time, the phase of the carrier is unique only in the range from $\phi = 0$ to $\phi = 2\pi$ or, equivalently, from $\phi = -\pi$ to $\phi = \pi$. Another form of representation, called the *phase trellis*, is obtained by plotting the phase trajectories within the bounds of modulo 2π, in the range $(-\pi, \pi)$. The phase trellis is best viewed as a three dimensional plot in which the quadrature components x_c and x_s appear on the surface of a cylinder of unit radius and are plotted against time. The

quadrature components are defined as $x_c(t;\mathbf{I}) = \cos\phi(t;\mathbf{I})$ and $x_s(t;\mathbf{I}) = \sin\phi(t;\mathbf{I})$. Figure 2.2-5 illustrates a phase cylinder for a binary CPM scheme with modulation index h = 2/3 and a raised cosine pulse of length 3T.

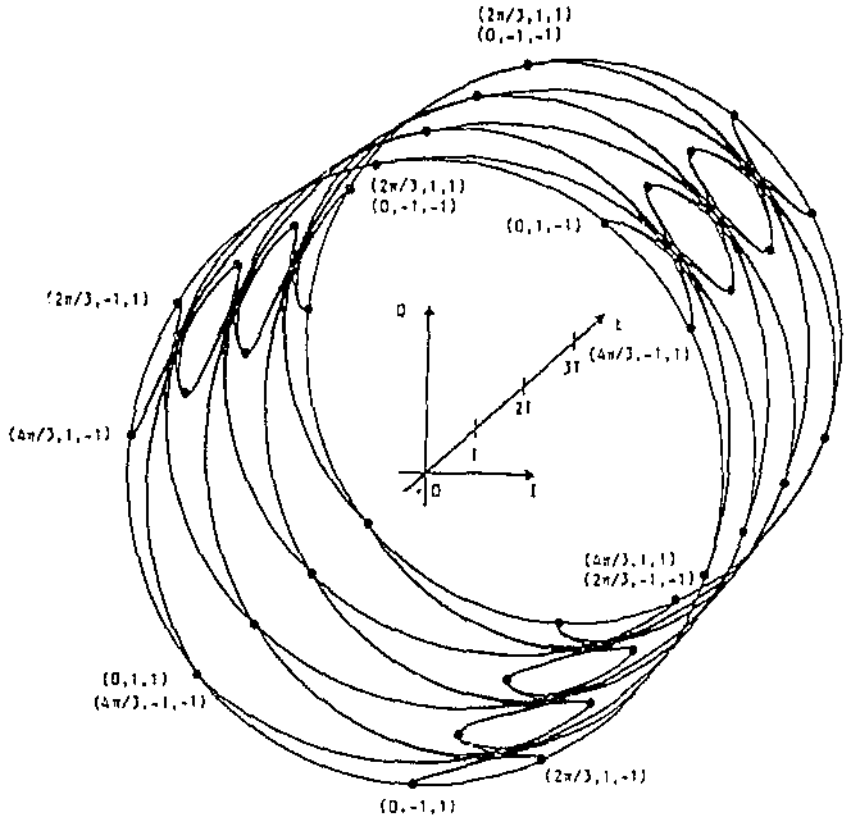

Figure 2.2-5: A phase cylinder for M=2, 3RC with h = 2/3 [1].

When the modulation index is defined as h = m / p, where m and p are relatively prime integers, a full response CPM signal at the time instants t = nT will have the following set of terminal phase states depending on whether m is even or odd.

Terminal phase states when m is even

$$\Theta_s = \left\{0, \frac{\pi m}{p}, \frac{2\pi m}{p}, \ldots, \frac{(p-1)\pi m}{p}\right\} \tag{2.2-3}$$

Terminal phase states when m is odd

$$\Theta_s = \left\{0, \frac{\pi m}{p}, \frac{2\pi m}{p}, \ldots, \frac{(2p-1)\pi m}{p}\right\} \tag{2.2-4}$$

From equations (2.2-3) and (2.2-4) it is apparent that there are p terminal phase states when m is even and 2p states when m is odd.

When the pulse shape extends beyond T as in partial response CPM, the number of terminal phase states is given as

$$N_S = \begin{cases} pM^{L-1} & \text{(even m)} \\ 2pM^{L-1} & \text{(odd m)} \end{cases} \qquad (2.2\text{-}5)$$

where M is the alphabet size. Note that with partial response CPM schemes, the number of phase states is increased by a factor of M^{L-1} and that the phase state transition from one state to another is not the actual phase trajectory.

2.3 Minimum-Shift Keying (MSK)

[8]

Minimum Shift Keying (MSK) is a special form of CPFSK, and hence CPM, where the modulation index is h = 1/2. The phase of the carrier in the interval $nT \le t \le (n+1)T$ is given by:

$$\begin{aligned} \phi(t;\mathbf{I}) &= \frac{1}{2}\pi \sum_{k=-\infty}^{n-1} I_k + \pi I_n q(t-nT) \\ &= \theta_n + \frac{1}{2}\pi I_n\left(\frac{t-nT}{T}\right), \qquad nT \le t \le (n+1)T \end{aligned} \qquad (2.3\text{-}1)$$

Therefore, the modulated carrier signal is

$$\begin{aligned} s(t) &= A\cos\left[2\pi f_c t + \theta_n + \frac{1}{2}\pi I_n\left(\frac{t-nT}{T}\right)\right] \\ &= A\cos\left[2\pi\left(f_c + \frac{1}{4T}I_n\right)t - \frac{1}{2}n\pi I_n + \theta_n\right], \qquad nT \le t \le (n+1)T \end{aligned} \qquad (2.3\text{-}2)$$

Expression (2.3-2) indicates that the MSK signal can be expressed as a sinusoid having one of two possible frequencies in the interval $nT \le t \le (n+1)T$. These two possible frequencies are:

$$\begin{aligned} f_1 &= f_c - \frac{1}{4T} \\ f_2 &= f_c + \frac{1}{4T} \end{aligned} \qquad (2.3\text{-}3)$$

Thus, expression (2.3-2) may now be written as:

$$s_i(t) = A\cos\left[2\pi f_i t + \theta_n + \frac{1}{2}n\pi(-1)^{i-1}\right], \qquad i = 1,2 \qquad (2.3\text{-}4)$$

The separation between the two frequencies f_1 and f_2 is 1/2T. This is the minimum frequency separation that is necessary to ensure the orthogonality of the signals $s_1(t)$ and $s_2(t)$ over a signalling interval of length T. This explains the naming of this modulation scheme as MSK.

2.4 Transmitter

[12]

This section presents the CPM transmitter as a conceptual model which shows how CPM can be transmitted based on the look-up table principle.

A conceptual general transmitter structure based on (2.1-3) is given in Figure 2.4-1. For multi-h CPM schemes, it is not practical to convert this structure into hardware because an exact relation between the symbol rate and the modulation index is required and this requires control circuitry.

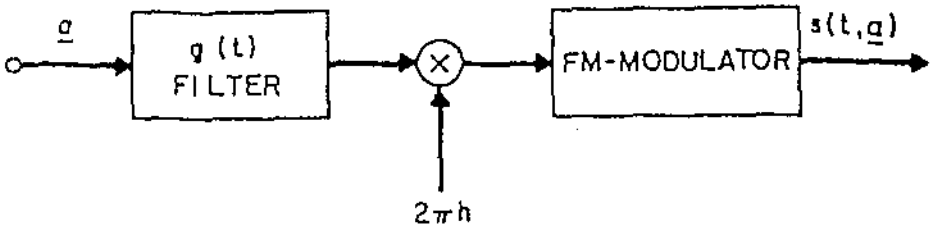

Figure 2.4-1: Conceptual model of a CPM modulator directly based on expression (2.3-1). Note that the symbol variable $\underline{\alpha}$ is equivalent to I in the notation used for this thesis [12].

Figure 2.4-2 below shows a CPM transmitter based on stored lookup tables. This approach is the most straight forward method of implementing a robust CPM transmitter. To understand this model, the CPM waveform is presented in its normalised form as

$$s_0(t) = s(t)/\sqrt{\frac{2E}{T}} = I(t)\cos(2\pi f_c t) - Q(t)\sin(2\pi f_c t) \qquad (2.4\text{-}1)$$

where the in-phase and quadrature components are $I(t) = \cos(\phi(t,\mathbf{I}) + \theta_n)$ and $Q(t) = \sin(\phi(t,\mathbf{I}) + \theta_n)$ respectively.

The ROM tables of the I(t) and Q(t) ROM blocks digitally store the in-phase and quadrature components of the CPM waveform for all possible modulation states. Each stored waveform spans a symbol interval and can be referenced by an address field. The data symbol sequence **I** is fed into the State ROM block where the modulation state is determined as S_n. Subsequently, S_n is routed to the I(t) and Q(t) ROM blocks and the corresponding signal waveform component for that state is sent to a digital-to-analogue converter (DAC). From the DAC, the in-phase and quadrature signal components are then modulated by multiplying them with the carrier components to finally generate the CPM signal. Note that S_n is also routed back to the State ROM module after a delay of one symbol interval. This is necessary because each new state is determined by the previous state and the current data symbol.

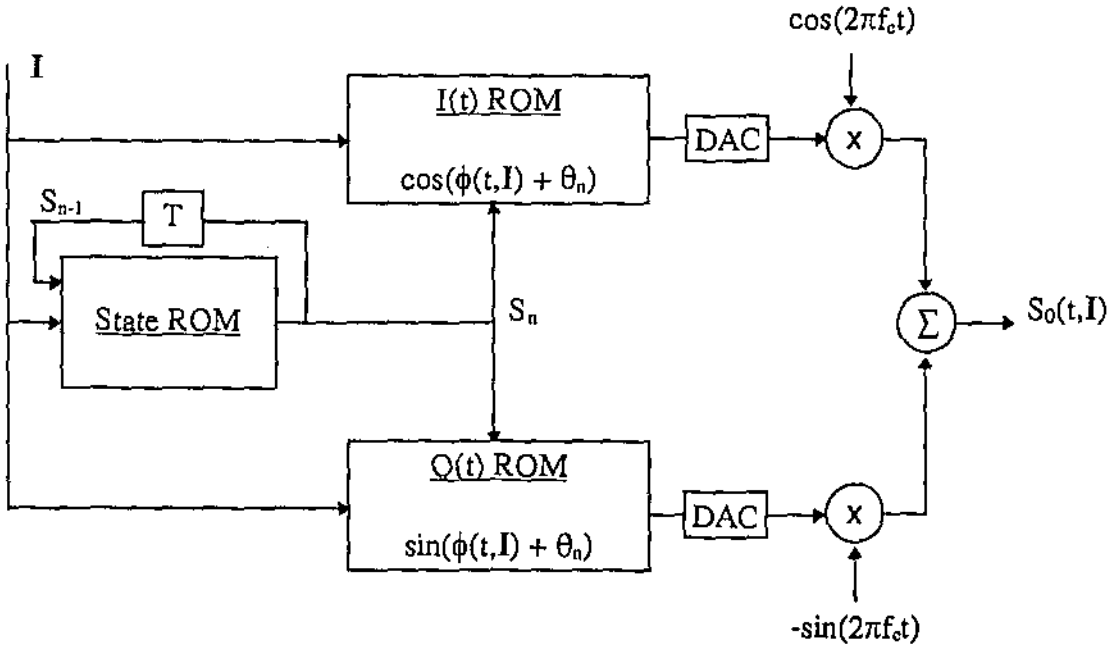

Figure 2.4-2: General CPM transmitter based on the look-up table principle [3].

2.5 Maximum Likelihood Sequence Reception

In this section, an optimum reception technique for signals with memory known as Maximum Likelihood Sequence (MLSE) reception is presented. A MLSE receiver is a receiver which selects the most likely signal sent from a set of possible signal waveforms, given a waveform r(t) which it has received [1]. This receiver is optimum if it chooses as the transmitted signal that which has the largest probability of being sent after observing the entire sequence of received signal r(t) [1] [10].

In the following part of this section, the decision rule for optimum MLSE reception is presented. Subsequently, the decision rule is adapted for an AWGN channel whereby an equivalent receiver called the MLSE *correlation receiver* is described.

If $s_i(t)$, for $i = 1,2,\ldots,M$, represents a set of all the possible transmitted signal waveforms and r(t) is defined as the observed signal waveform at the receiver, an optimum receiver sets the message estimate to s_k if for $i = 1,2,\ldots,M$, the following is satisfied [1]

$$f(s_k(t)|r(t)) > f(s_i(t)|r(t)) \qquad i \neq k \tag{2.5-1}$$

where f denotes a probability density function. Equation (2.5-1) defines the optimum receiver's decision rule. The probabilities indicated are called *a posteriori* because they relate to message probabilities after a specific waveform has been received. Hence, a receiver which relies on equation (2.5-1) is also known as a *maximum a posteriori* (MAP) receiver [10].

Using Bayes' theorem, it can be shown that [10]

$$f(s_i(t)|r(t)) = \frac{f_R(r(t)|s_i(t))P(s_i(t))}{f_R(r(t))} \tag{2.5-2}$$

where $f_R(r(t)|s_i(t))$ is the conditional joint probability density of the random variables defining r(t), $f_R(r(t))$ is the probability density without condition, and $P(s_i(t))$ is the source probability. On substituting (2.5-2) into (2.5-1), the MAP decision rule becomes [10]:

$$f_R(r(t)|s_k(t))P(s_k(t)) > f_R(r(t)|s_i(t))P(s_i(t)) \qquad (2.5\text{-}3)$$

When all messages are equally probable, probabilities $P(s_k(t))$ and $P(s_i(t))$ may be disregarded such that [10]:

$$f_R(r(t)|s_k(t)) > f_R(r(t)|s_i(t)) \qquad (2.5\text{-}4)$$

The receiver that is based on (2.5-4) is called a *MLSE receiver* [10].

For an additive white Gaussian noise (AWGN) channel, it is shown in [1] that the left hand side of expression (2.5-4) can be expressed as a multivariate density function of r_1, r_2,..., r_{J_s}. This is conditioned on the transmitted signal $s_i(t)$ where r_1, r_2,..., r_{J_s} and s_{i1}, s_{i2}, ..., s_{iJ_s} are the expansions of r(t) and $s_i(t)$, respectively, to span a J_s-dimensional signal space. The elaborate analysis involved is not provided but the final expression is [1]:

$$f_R(r_1,\ldots,r_{J_s}|s_i(t)) = \prod_{j=1}^{J_s} \frac{\exp[-(r_j - s_{ij})^2 / N_0]}{(\pi N_0)^{1/2}}$$

$$= \frac{\exp\left[-\sum_{j=1}^{J_s}(r_j - s_{ij})^2 / N_0\right]}{(\pi N_0)^{J_s/2}} \qquad (2.5\text{-}5)$$

Hence, the maximum likelihood receiver finds the maximum of (2.5-5) over the choice of $s_i(t)$. It can be observed from expression (2.5-5) that the value of the density depends on only one quantity that relates to the signal set. This quantity is $\sum_{j=1}(r_j - s_{ij})^2$. In the normed vector space that was set up, Parseval's identity [1] can be used to show that this expression equals $\int[r(t) - s_i(t)]^2 dt$. This is the square norm of r(t) - $s_i(t)$ [1], or otherwise known as the *square Euclidean distance* between the signals, r(t) and $s_i(t)$. It has now been shown that the sum-of-components form of the distance is also a sum of squares of the differences in a set of orthogonal directions. This distance is analogous to distance in the real world. With this understanding, it is apparent that the density f_R of expression (2.5-5) is maximised by choosing $s_i(t)$ such that the distance between $s_i(t)$ and r(t) is minimal. Essentially, this can be achieved by finding the minimum of

$$\int [r(t) - s_i(t)]^2 dt = \int r^2(t)dt + \int s_i^2(t)dt - 2\int r(t)s_i(t)dt \qquad (2.5\text{-}6)$$

for each i [1]. The first term of expression (2.5-6) is a constant with respect to i. Hence, only the second and third terms need to be determined by the receiver. These terms are the energy of $s_i(t)$, and the cross correlation of r(t) and $s_i(t)$ respectively. If all the transmitted signals are of equal energy, then the calculation of the second term may be avoided [1].

In summary, the objective of an optimum MLSE receiver of an AWGN channel [1] [10] is:

To determine k such that $s_k(t)$ results in the largest cross correlation with r(t).

A receiver based on these calculations is known as a MLSE correlation receiver [1].

2.6 Optimum Receiver for CPM Signals

The optimum MLSE receiver for CPM signals consists of a cross correlator followed by a MLSE detector [1]. The task of the MLSE detector is to search the paths through the state trellis for the minimum Euclidean distance path. This method of detection is optimum because the most likely estimate of the signal sequence is obtained [1]. An efficient technique, known as the Viterbi algorithm, is employed to perform this search. The following sub-sections present the *state trellis* structure for CPM signals which is necessary to effectively detect CPM signals; the *cross correlation metric computations* which is required for performing realtime cross correlation; and the *Viterbi algorithm* which is an efficient method of performing MLSE detection in realtime.

2.6.1 State Trellis Structure for CPM

[8]

The concept of states and state traversals of CPM signals are necessary to detect CPM signals effectively. By understanding these concepts, it is then possible to generate the

possible signal waveforms (for the possible state transitions) which are used by the cross correlator and subsequently the MLSE detector to determine the most likely signal waveform sent.

For a fixed modulation index h, equation (2.2-1) may be expressed as

$$\begin{aligned}\phi(t;I) &= 2\pi h \sum_{k=-\infty}^{n} I_k q(t-kT) \\ &= \pi h \sum_{k=-\infty}^{n-L} I_k + 2\pi h \sum_{k=n-L+1}^{n} I_k q(t-kT) \\ &= \theta_n + \phi(t;\mathbf{I}), \qquad nT \le t \le (n+1)T \end{aligned} \tag{2.6.1-1}$$

where it is assumed that q(t) = 0 for t < 0, q(t) = 1/2 for t ≥ LT, and

$$q(t) = \int_0^t g(\tau)d\tau \tag{2.6.1-2}$$

The signal pulse g(t) = 0 for t < 0 and t ≥ LT. If L = 1 it is a full response CPM signal. Otherwise, if L > 1 where L is a positive integer, it is a partial response CPM signal.

When h is rational, such that h = m/p where m and p are relatively prime positive integers, the CPM scheme can be represented as a trellis. This trellis may have either p or 2p phase states depending on whether m is even or odd respectively. Expressions (2.2-3) and (2.2-4) are restated below.

Terminal phase states when m is even

$$\Theta_s = \left\{0, \frac{\pi m}{p}, \frac{2\pi m}{p}, \ldots, \frac{(p-1)\pi m}{p}\right\} \tag{2.2-3}$$

Terminal phase states when m is odd

$$\Theta_s = \left\{0, \frac{\pi m}{p}, \frac{2\pi m}{p}, \ldots, \frac{(2p-1)\pi m}{p}\right\} \tag{2.2-4}$$

These are the only states in the state trellis when L = 1. However, if L > 1, there exists an additional number of states due to the partial response character of the signal pulse g(t). In order to identify the additional states, expression (2.6.1-1) is re-expressed as:

$$\phi(t;\mathbf{I}) = 2\pi h \sum_{k=n-L+1}^{n-1} I_k q(t-kT) + 2\pi h I_n q(t-nT) \qquad (2.6.1\text{-}3)$$

The first term on the right-hand side of expression (2.6.1-3) is dependent on the information symbols (I_{n-1}, I_{n-2}, ..., I_{n-L+1}) which is called the *correlative state* vector. This represents the phase term corresponding to signal pulses that have not reached their final value. The second term represents the phase contribution due to the most recent symbol I_n. Thus, the state of the CPM signal (and the modulator) at time t = nT is expressed as the combined *phase state* and *correlative state* for a partial response signal pulse of length LT, where L > 1.

This is represented as:

$$S_n = \{\theta_n, I_{n-1}, I_{n-2}, ..., I_{n-L+1}\} \qquad (2.6.1\text{-}4)$$

Consequently, the total number of states for partial response CPM when h = m/p is:

$$N_S = \begin{cases} pM^{L-1} & \text{(even m)} \\ 2pM^{L-1} & \text{(odd m)} \end{cases} \qquad (2.6.1\text{-}5)$$

The reception of another symbol in the successive symbol interval will result in a transition of states. Assume that the current state of the signal at t = nT is S_n. After receiving the next symbol in the time interval $nT \le t \le (n+1)T$, the new state of the modulator will be S_{n+1}. The new state at t = (n+1)T can be represented as

$$S_{n+1} = (\theta_{n+1}, I_n, I_{n+1}, ..., I_{n-L+2}) \qquad (2.6.1\text{-}6)$$

where the new phase state is

$$\theta_{n+1} = \theta_n + \pi h I_{n-L+1} \qquad (2.6.1\text{-}7)$$

An example illustrating a state transition trellis is provided in Figure 2.6.1-1 for the binary 3RC scheme with h = 4/5.

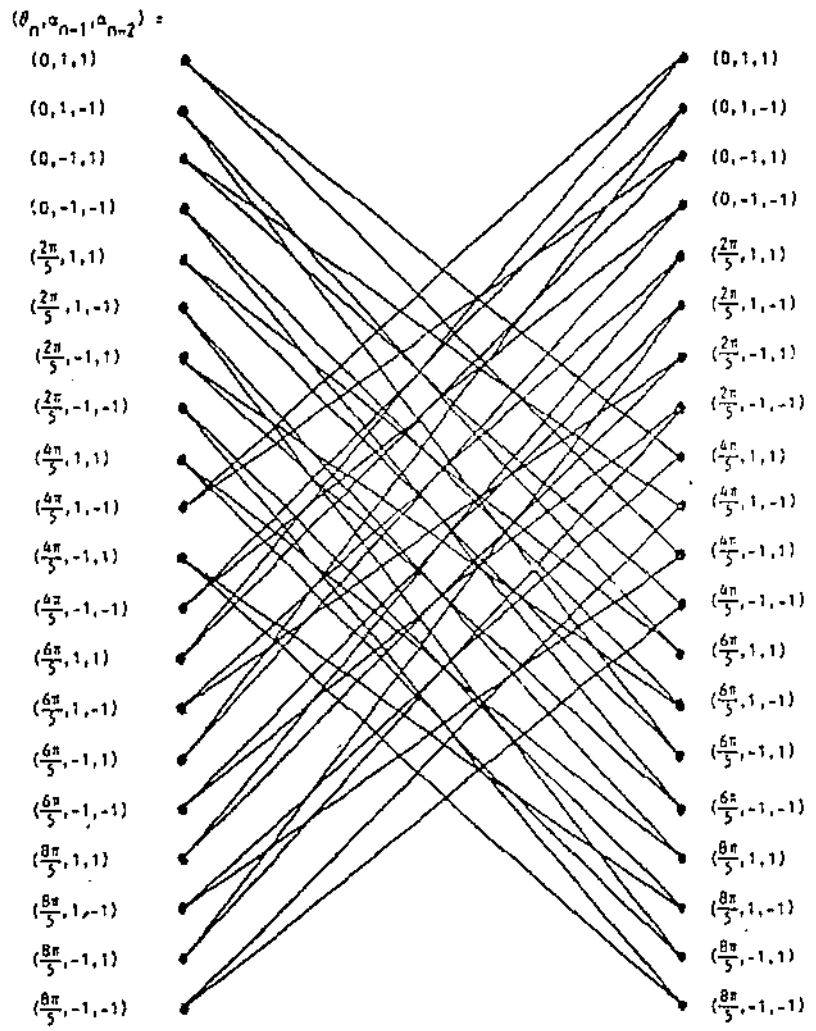

Figure 2.6.1-1: The state trellis diagram for the binary 3RC scheme with h = 4/5. Note that the symbol variable $\underline{\alpha}$ is equivalent to I in the notation used for this thesis [1].

Below, Figure 2.6.1-2 illustrates a path through the state trellis for the binary 2REC scheme with h = 3/4. This path corresponds to the sequence (1, -1, -1, -1, 1, 1) which was received.

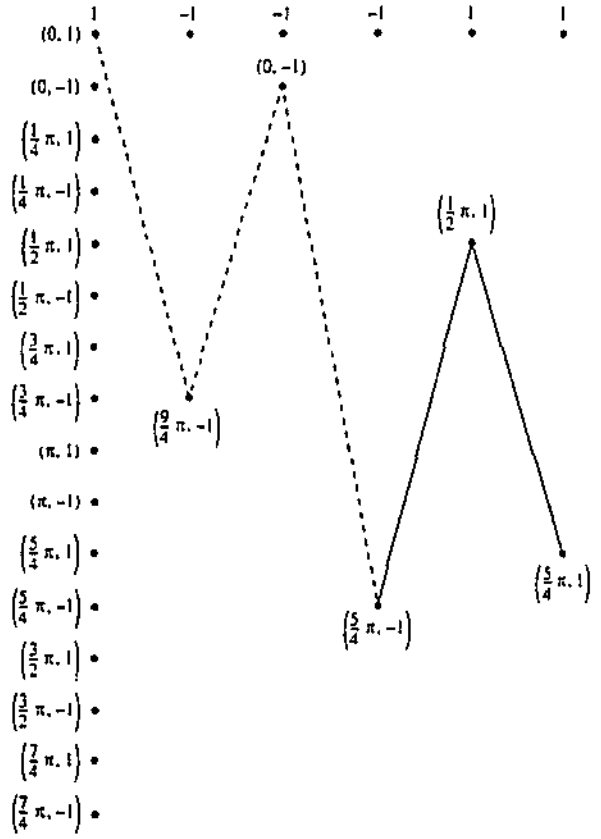

Figure 2.6.1-2: A single path through the state trellis for the binary 2REC scheme with h = 3/4 corresponding to the sequence (1, -1, -1, -1, 1, 1) [8].

2.6.2 Correlation Metric Computation

From the developments in section 2.5, expression (2.5-6) shows that the MLSE receiver minimises

$$\int_a^b [r(t) - s_i(t)]^2 dt \qquad (2.6.2\text{-}1)$$

with respect to an estimated sequence of symbols $\hat{s}(t)$ over the entire interval of the received signal r(t), which may be any symbol intervals long, spanning [a, b].

When all the transmitted signals are of equal energy, it is equivalent to maximising the correlation [1]:

$$\int_a^b r(t)s_i(t)dt \qquad (2.6.2\text{-}2)$$

In principal, the correlation receiver of expression (2.6.2-2) cross correlates all possible transmitted signals s(t, I) with the received signal and the maximising signal sequence $\hat{s}(t)$ is chosen as the received signal. The information bearing data sequence can then be deduced knowing which of the possible signals was received.

In practice it is not possible to implement a correlation receiver based on expression (2.6.2-2) even if the received sequence is not infinitely long [1]. It is not a feasible structure in practice because a large memory capacity is required to store the received signal waveform and all the possible signal waveforms which are to be correlated later. Also, realtime processing cannot be facilitated directly from (2.6.2-2) because its derivation cannot be completed until at least the last signal waveform corresponding to the last symbol is received. This could take a very long time for long sequences.

A possible solution to this problem is to use a *sliding window* variant of the MLSE detection technique [1]. Since this variant technique considers only a sliding window length of the signal sequence for its decision making, and not the entire signal sequence, it is sub-optimum [1]. The proposed sliding window method for MLSE detection involves limiting the time range considered by the correlation receiver. At every instant, while the detector is still receiving r(t), the modified detector chooses the most likely signal sequence from a set of possible signal sequences over the last N_t symbol intervals. The choice is made using a slightly modified expression (2.6.2-2) where the limits of integration spans the last N_t symbol intervals. Hence, the region under consideration (the window) is always limited to N_t symbol periods and it slides forward with every successive symbol received. When the most likely signal sequence is determined, the data sequence can be deduced, perhaps by the assistance of a lookup table mapping the possible modulated signal sequences to the corresponding data. Compared with the non-windowed MLSE detector, the length of r(t) and possible signal sequences which the

sliding window detector needs to remember is bounded to N_t symbol intervals. This approach will reduce the memory requirement of the receiver especially if r(t) is very long. It is now evident that this modification allows the receiver to operate in real time and with finite memory requirements.

The memory requirement of the sliding window MLSE receiver can be further reduced by employing *correlation metric computations* which will be described later in this section. Essentially, this technique requires that sub-correlations be performed over each symbol interval as r(t) is being received rather than over the last N_t symbol intervals. The metrics are stored cumulatively for all the possible paths spanning the trellis bounded by the sliding window. The MLSE decision is then made by choosing the signal sequence from the set of possible sequences spanning the bounded trellis which has the largest cumulative correlation. Using correlation metric computations, the length of r(t) and that of all the possible signal sequences which the detector needs to remember is now bounded to one symbol interval.

It is important to note that the practical MLSE receivers proposed above are sub-optimal. This is because an optimum MLSE receiver in AWGN channel is based on expression (2.6.2-2) and calculates the correlation over the entire interval of the received signal r(t), which may be any number of symbol intervals long. With the practical receivers proposed, it is not feasible to consider the correlation for the entire length of r(t) which may be infinitely long because of the infinite memory requirement and infinite delay in processing.

The correlation metrics (CM) is defined by [8]:

$$CM_n(\mathbf{I}) = \int_{-\infty}^{(n+1)T} r(t)\cos[w_c t + \phi(t;\mathbf{I})]dt$$

$$= CM_{n-1}(\mathbf{I}) + \int_{nT}^{(n+1)T} r(t)\cos[w_c t + \phi(t;\mathbf{I}) + \theta_n]dt \qquad (2.6.2\text{-}3)$$

The term $CM_{n-1}(\mathbf{I})$ represents the cumulative metrics for the sequences up to time nT while expression (2.6.2-4) below represents the additional metric increments contributed by the signal in the time interval $nT \le t \le (n+1)T$.

$$\upsilon_n(I;\theta_n) = \int_{nT}^{(n+1)T} r(t)\cos[w_c t + \phi(t;I) + \theta_n]dt \qquad (2.6.2\text{-}4)$$

While a sliding window, MLSE detector utilising CM computations is more memory efficient than its original form, it is computationally expensive for large sliding windows. The length of the sliding window and the number of possible signal sequences to be considered during each symbol interval is related exponentially. As the length of the sliding window, N_tT is increased, the number of possible paths in the bounded state trellis increases exponentially. It is quite evident that a longer sliding window will improve signal detection just as a MLSE detector with an infinitely long window is optimum. However, as mentioned earlier, any efforts to increase the window size will result in slower processing.

As an example, consider a CPM scheme with M = 8 using the described MLSE detector with sliding window length $N_t = 5$. Since the number of possible paths in a bounded state trellis is M^{N_t}, the number of possible paths for this example is 32,768. This is the number of possibilities that need to be considered during each symbol interval when detecting r(t). Using a special technique for MLSE detection called the Viterbi algorithm, this Figure can be dramatically reduced to 256, as described in the next section.

2.6.3 Viterbi Algorithm

A definition of the Viterbi algorithm from [1] is as follows:

"The Viterbi algorithm (VA) is a recursive optimal solution to the problem of estimating the state sequence of a discrete time finite state Markov process observed in memoryless noise".

The VA is commonly used for the decoding of convolutional codes but it can be adapted for MLSE sequence estimation so that it is used to choose the signal sequence that maximises expression (2.6.2-2) [1]. Since a state transition maps uniquely to a data sequence, a maximum likelihood estimate of the data sequence is also obtained. In its

basic form, the Viterbi algorithm is optimum in the sense that the most likely estimate of the sequence of states is always obtained [15].

The VA in its basic form is not feasible for the same reasons as an optimum MLSE sequence receiver is not feasible in practice. For the detection of CPM schemes, a modified sliding window VA, utilising correlation metric (CM) computations is proposed.

The reception of CPM signals using MLSE sequence estimation involves correlating the received signal r(t) with all the possible signal waveforms over the current symbol interval. This is performed by the incremental CM. The CMs are subsequently processed by the VA and selected values are stored away for as long as the sliding window length, N_tT. The number of possible signal waveforms is equivalent to the number of state trellis transitions during each symbol interval. Since each state in the trellis has M possible state transitions, the total number of state transitions is pM^L (or $2pM^L$ when m is odd as shown by (2.6.1-5)) for partial response CPM, and pM (or 2pM) for full response CPM; where, pM^{L-1} (or $2pM^{L-1}$) is the total number of states for the partial response scheme, and p (or 2p) is the number of phase states for the full response scheme. This implies at every symbol interval, that the receiver must calculate pM^L (or $2pM^L$) CMs for partial response CPM and pM (or 2pM) CMs for full response CPM [8].

At the initial stages of reception, the Viterbi detector accumulates the CMs until a memory buffer of size N_{ST} x N_t is filled, where N_{ST} is the number of state transitions in the state trellis. This memory buffer is called the *sliding window metric buffer*. This is a *first-in-first-out* (FIFO) buffer where the correlation metrics of successive symbol intervals are fed in. Once filled, it is possible to determine the *most likely windowed path* (MLWP) which has the largest cumulative CM spanning the windowed trellis. As the window slides forth with successive symbol intervals, it is possible to determine a state transition trajectory from the last state of each successive MLWP. The state transition trajectory is then used to deduce the received data sequence. This is normally performed by using a lookup table in ROM which stores the mapping between state transitions and data symbols.

As larger window sizes are used, the number of possible paths within the windowed trellis increases exponentially. To minimise the number of possible paths to be considered at each symbol interval, *surviving windowed paths* (SWPs) are used.

The SWP of each final state in a windowed trellis is defined as the windowed path arriving at that final state with the largest cumulative CMs. Note that the number of SWPs is always equal to the number of states on one side of the state trellis diagram. When m is even, this is equal to p for a full response CPM scheme or pM^{L-1} for a partial response CPM scheme. Otherwise, when m is odd, it is equal to 2p for a full response CPM scheme or $2pM^{L-1}$ for a partial response CPM scheme. With each successive symbol interval, the number of SWPs is extended by a factor of M to account for all the possible state transitions. The new SWPs will then be chosen from the extended SWPs (ESWP) such that the SWP for each new state is the ESWP with the largest cumulative CMs. In turn, the MLWP for the entire trellis will be chosen from the new SWPs such that it is the SWP with the largest cumulative CMs. This process is repeated for successive symbol intervals [8].

Using the VA, the number of ESWPs which need to be considered during each symbol interval is equal to the number of state transitions multiplied by M [8]. Consider a partial response CPM scheme where $N_t = 5$, $p = 4$, $M = 8$ and $L = 2$. The number of ESWPs to be considered by the sliding window VA detector at each symbol interval is $pM^L = 256$. It was determined that the number of paths an equivalent sliding window MLSE detector has to consider is $M^{N_t} = 32{,}768$. Evidently, the complexity of the sliding window VA detector is independent of N_t, unlike the sliding window MLSE detector, suggesting that N_t should be selected as wide as possible, providing the processing delay can be tolerated and the memory requirement of the detector is available. It is apparent that for demanding CPM schemes, the sliding window MLSE detector should be replaced by the sliding window VA detector for improved performance.

2.7 Physical Realisation of a Sub-Optimum Receiver

[1]

The physical implementation of a MLSE receiver utilising VA detection is presented in this section. This receiver is comprised of a quadrature receiver as shown in Figure 2.7-1 below, followed by an array of correlation filters. Subsequently, the output of these correlation filters are in turn fed into a Viterbi processor to obtain the received signals.

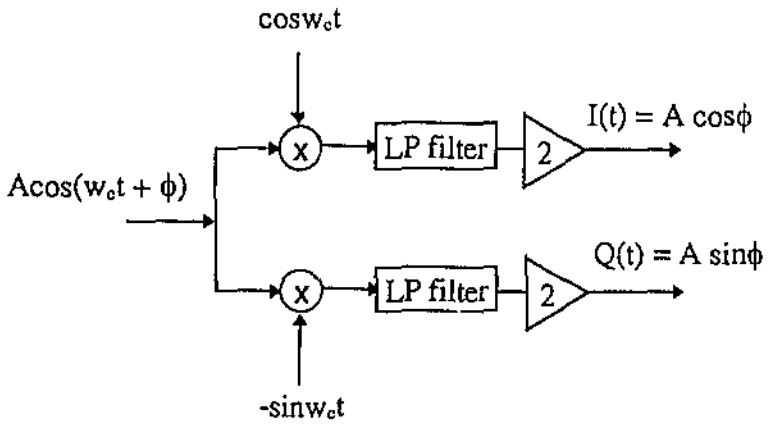

Figure 2.7-1 illustrates a quadrature receiver. The purpose of the quadrature receiver is to separate the bandpass signal into its in-phase I(t) and quadrature Q(t) components where the bandpass signal $A\cos(w_c t + \phi) = A\cos\phi \cos w_c t - A\sin\phi \sin w_c t = I(t) \cos w_c t - Q(t) \sin w_c t$.

Expression (2.6.2-4) for the CM is recalled below:

$$\upsilon_n(\mathbf{I};\theta_n) = \int_{nT}^{(n+1)T} r(t)\cos[w_c t + \phi(t;\mathbf{I}) + \theta_n]dt \qquad (2.6.2\text{-}4)$$

The metric $\upsilon_n(\mathbf{I};\theta_n)$ is obtained by feeding the signal r(t) into a filter and sampling the output of the filter at t = (n+1)T. This requires a bank of filters to be used. Essentially, the receiver correlates the received signal over one symbol interval with all possible transmitted alternatives over the symbol interval.

Assuming additive noise n(t) defined in the bandpass form

$$n(t) = x(t) \cos w_c t - y(t) \sin w_c t \qquad (2.7\text{-}1)$$

is added in the channel, such that:

$$\hat{r}(t) = I(t) \cos(w_c t) - Q(t) \sin(w_c t) + n(t) \qquad (2.7\text{-}2)$$

the output of the quadrature receiver components are:

$$\hat{I}(t) = \sqrt{\frac{2E}{T}} I(t) + x(t)$$
$$\hat{Q}(t) = \sqrt{\frac{2E}{T}} Q(t) + y(t) \qquad (2.7\text{-}3)$$

Equation (2.7-4) below is obtained by inserting (2.7-2) into (2.6.2-4) and omitting double frequency terms due to the low pass filters of the quadrature receiver.

$$\upsilon_{\theta_n}(\mathbf{I}) = \cos(\theta_n)\int_{nT}^{(n+1)T} \hat{I}(t)\cos[\phi(t,\mathbf{I})]dt + \cos(\theta_n)\int_{nT}^{(n+1)T} \hat{Q}(t)\sin[\phi(t,\mathbf{I})]dt$$
$$+\sin(\theta_n)\int_{nT}^{(n+1)T} \hat{Q}(t)\cos[\phi(t,\mathbf{I})]dt - \sin(\theta_n)\int_{nT}^{(n+1)T} \hat{I}(t)\sin[\phi(t,\mathbf{I})]dt \qquad (2.7\text{-}4)$$

Figure 2.7-2 below illustrates the receiver, emphasising the operation of the quadrature receiver.

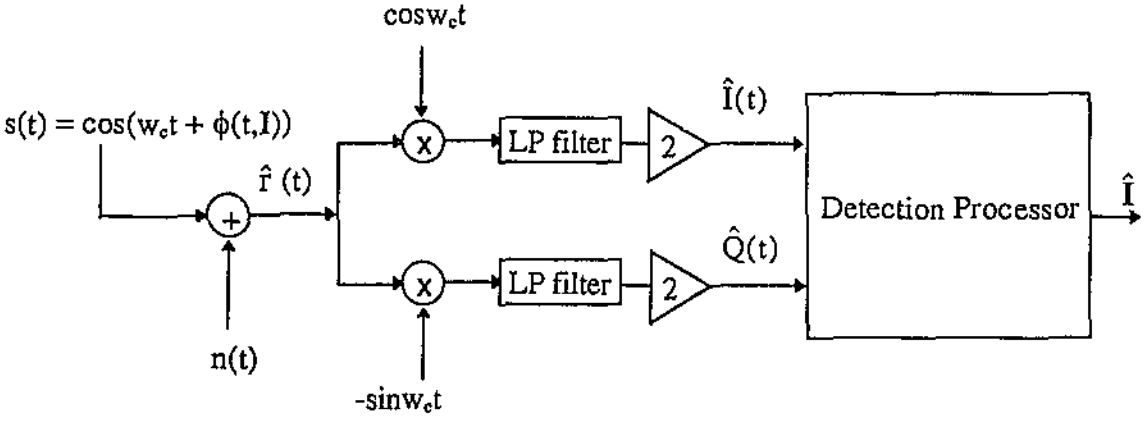

Figure 2.7-2: The quadrature receiver and detection processor.

Expression (2.7-4) can be interpreted as $4M^L$ baseband filters with the impulse responses:

$$h_c(t,\mathbf{I}) = \begin{cases} \cos\left[2\pi h \sum_{j=-L+1}^{0} I_j q((1-j)T-t)\right] \\ 0 \qquad \text{for } t \text{ outside } [0,T] \end{cases}$$

$$h_s(t,\mathbf{I}) = \begin{cases} \sin\left[2\pi h \sum_{j=-L+1}^{0} I_j q((1-j)T-t)\right] \\ 0 \qquad \text{for } t \text{ outside } [0,T] \end{cases} \qquad (2.7\text{-}5)$$

Since every I-sequence has a corresponding sequence with reversed sign, the number of filters required can be reduced by a factor of 2. Figure 2.7-3 below shows a receiver with $2M^L$ matched filters. The outputs of these filters are sampled once every symbol interval, which produces the metrics $\upsilon_{\theta_n}(\mathbf{I})$.

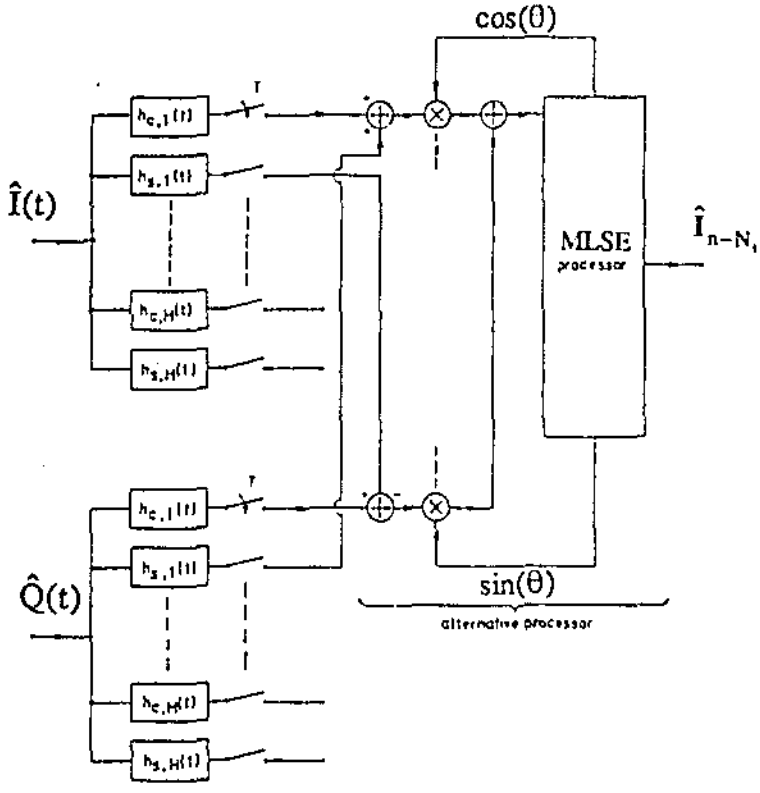

Figure 2.7-3: Receiver with baseband filter bank [1].

2.8 Performance of CPM Signals

The objective of this section is to understand the bit error performance behaviour expected of CPM schemes. In particular, the effect of varying parameters such as the alphabet size M, the modulation index h, and the length of the transmitted pulse on bit error performance is characterised.

As stated in [8], the error rate performance for CPM is dominated by the term corresponding to the minimum Euclidean distance and this can be expressed as

$$P_M = K_{\delta_{min}} Q\left(\sqrt{\frac{E_b}{N_0}\delta^2_{min}}\right) \qquad (2.8\text{-}1)$$

where

$$\delta^2_{min} = \lim_{N\to\infty} \min_{i,j} \delta^2_{ij} \qquad (2.8\text{-}2)$$

It is apparent from (2.8-1) that the larger δ^2_{min} is for a CPM scheme, the better its bit error rate performance. Expression (2.8-2) shows that the minimum Euclidean distance δ^2_{min} can be obtained by finding the minimum distance between any two possible paths over an infinitely long interval. Suppose there are two signals $s_i(t)$ and $s_j(t)$ corresponding to two phase trajectories $\phi(t,\mathbf{I}_i)$ and $\phi(t,\mathbf{I}_j)$ where the sequences $\mathbf{I}_i$ and $\mathbf{I}_j$ are different in their first symbol. The minimum Euclidean distance for these two signals over an interval NT is defined as [8]:

$$d_{ij}^2 = \int_0^{NT} [s_i(t) - s_j(t)]^2 dt \tag{2.8-3}$$

After some mathematical development [8], which is not provided, expression (2.8-3) can be expressed as

$$d_{ij}^2 = \frac{2E}{T} \int_0^{NT} \{1 - \cos[\phi(t;\mathbf{I}_i) - \phi(t;\mathbf{I}_j)]\} dt \tag{2.8-4}$$

where E is the signal energy. To express d_{ij}^2 in terms of the bit energy E_b, (2.8-4) may be expressed as

$$d_{ij}^2 = 2E_b \delta_{ij}^2 \tag{2.8-5}$$

where

$$E = E_b \log_2 M \tag{2.8-6}$$

$$\delta_{ij}^2 = \frac{\log_2 M}{T} \int_0^{NT} [1 - \cos\phi(t,\xi)] dt \tag{2.8-7}$$

The term ξ is defined as $\xi = \mathbf{I}_i - \mathbf{I}_j$ and any element of ξ can take the values 0, ±2, ±4, ±... ±2(M-1), except that $\xi_0 \neq 0$. Substituting (2.8-7) into (2.8-2) the following is obtained [8]:

$$\delta_{min}^2 = \lim_{N \to \infty} \min_{i,j} \left\{ \frac{\log_2 M}{T} \int_0^{NT} [1 - \cos\phi(t,\xi)] dt \right\} \tag{2.8-8}$$

Since δ^2_{min} characterises the performance of CPM with MLSE sequence estimation, it is possible to investigate the effect on δ^2_{min} resulting from varying M, h, and L in partial response CPM.

Typically, the first two paths to merge from a set of possible paths will provide the minimum Euclidean distance [8]. The merging of two possible paths occur when the two paths are initially dissimilar up to a certain point in time, beyond which, the two paths are exactly identical indefinitely. In order to determine the minimum Euclidean distance for a particular CPM scheme, the sequences providing the first merge must be determined. Practically, this is achieved by observation for simple schemes, while for more complex schemes, an exhaustive computer search is performed [1]. Once the two paths are determined, an expression for the minimum Euclidean distance can be obtained. This expression defines the upper bound on the minimum Euclidean distance [8].

As an example [8], consider the binary full response CPM scheme. The following sequences can be observed.

$$\mathbf{I}_i = +1, -1, I_2, I_3$$
$$\mathbf{I}_j = -1, +1, I_2, I_3 \qquad (2.8\text{-}9)$$

Note that the sequences merge after the second symbol. This corresponds to the difference sequence:

$$\xi = \{2, -2, 0, 0, \ldots\} \qquad (2.8\text{-}10)$$

The Euclidean distance for this sequence is calculated from (2.8-7) and is equal to:

$$d_B^2(h) = 2\left(1 - \frac{\sin 2\pi h}{2\pi h}\right), \qquad M = 2 \qquad (2.8\text{-}11)$$

Expression (2.8-11) provides the upper bound for M = 2. For example, where h = 1/2, which corresponds to MSK, we have $d^2_B(1/2) = 2$, so that $\delta^2_{min}(1/2) \leq 2$. The reason why it is an 'upper bound' is because for certain values of h, it may not be possible to attain the upper bound distance.

Figure 2.8-1 shows the plots of $d^2_B(h)$ versus h for various value of M. It is evident from this Figure that large gains in performance can be achieved by increasing the alphabet size M. It must be remembered that the upper bound may not be achievable for all values of h ($\delta^2_{min}(h) \leq d^2_B(h)$).

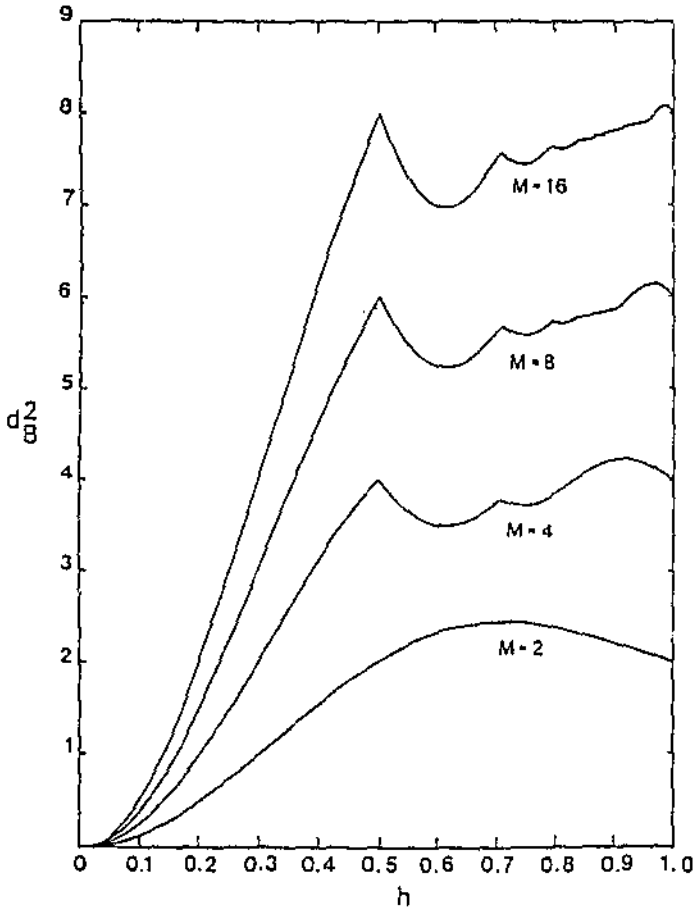

Figure 2.8-1: The upper bound d^2_B as a function of h for M-ary 1REC modulations, M = 2, 4, 8, 16 [1].

Figure 2.8-2 illustrates that significant performance gains can also be obtained by using partial response CPM schemes. As shown by the plots, increasing the pulse length L for raised cosine pulse shaped quaternary CPM can dramatically increase the minimum Euclidean distance.

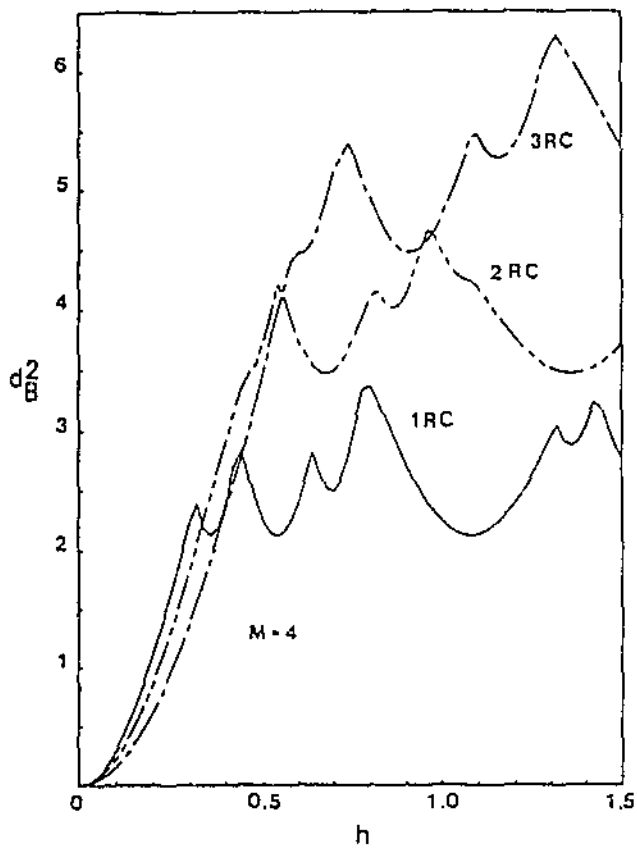

Figure 2.8-2: The upper bounds d^2_B for quaternary schemes 1RC to 3RC [1].

Consider the influence of observation width N on the performance of CPM schemes. Using (2.8-8) as the basis of their analysis, it is shown in [1] that the upper bound on the minimum Euclidean distance can be achieved when the observation length $N \geq N_B$, where N_B is the minimum observation length required to attain the upper bound. By setting the observation length N beyond N_B, it is not possible to attain a better performance in a noise-free ideal channel [1].

The influence of N on the performance of CPM schemes is best illustrated by Figure 2.8-3. For the quaternary 3RC scheme, $N_B = 12$. The plots show that as N is increased progressively, the Euclidean distance of the scheme tends closer to the upper bound. Also evident, is that the upper bound is reached for $N \geq 12$. It is worth noting that the Euclidean distance is at a minima at around $h = 0.7$ and $h = 1$ for N in the range of [6,12]. These values of h are called *weak modulation indices* and should be avoided when $N < N_B$ [8].

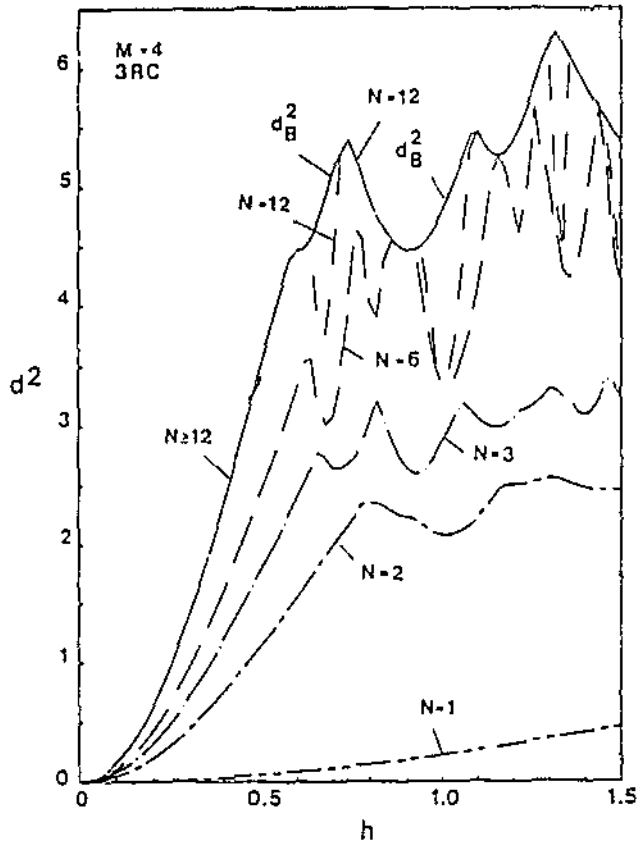

Figure 2.8-3: The minimum distance vs. h for quaternary 3RC with N = 1, 2, 3, 4, and 12 interval observation. N_B = 12 for h in the region (0, 0.5) [1].

The choice of sliding window length N_t to result in an error free performance in a noise-free ideal channel is very much related to N_B. The path memory N_t should be selected as large as N_B, and experiment has shown that N_B is almost always enough [1].

The spectral performance of CPM schemes is also influenced by the various parameters of modulation. Increased h results in a modulated signal with a larger bandwidth occupation while increased L implies a smaller bandwidth [8]. Figure 2.8-4 shows that as h is progressively increased, the corresponding power spectral density increases also. Figure 2.8-5 shows the influence of L on the power spectral density of a binary RC scheme with h = 1/2 for various pulse lengths L. Evident from these plots is that as L is increased from the plots of 1RC through to 5RC, the bandwidth is gradually reduced.

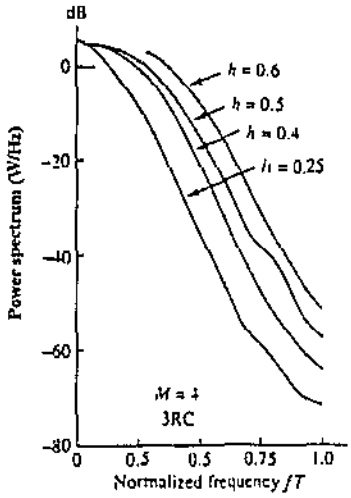

Figure 2.8-4: Power density spectrum for M = 4 CPM with 3RC and different modulation indices [8].

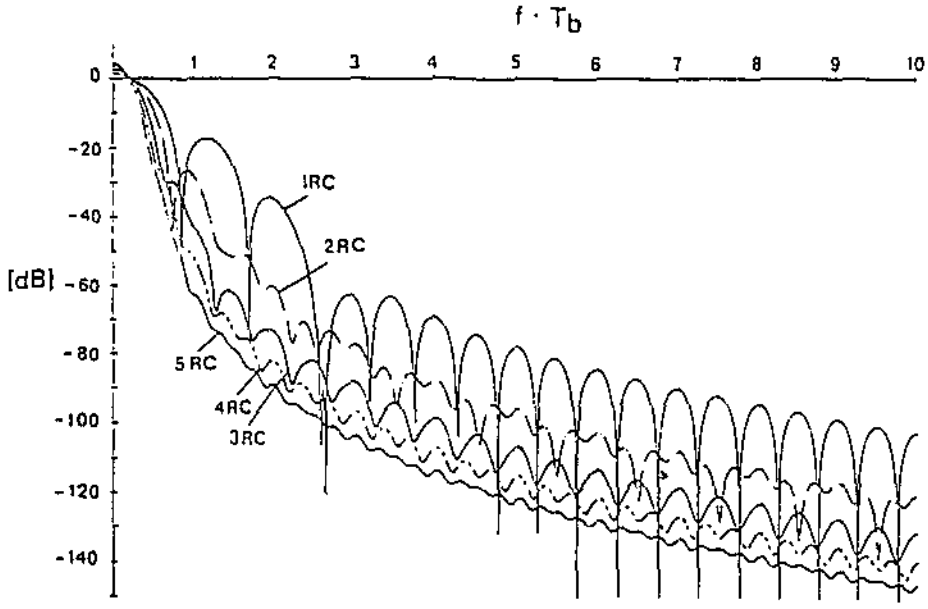

Figure 2.8-5: Power spectra for binary RC schemes when h = 1/2, showing 1RC through to 5RC with equiprobable data [1].

The pulse waveform, g(t) can be shaped to improve the bandwidth performance of the modulated signal [12]. Various pulse shapes are shown in Table 2.2-1, which can be used to obtain a narrower spectral occupancy.

While increasing M, L and N_t is likely to improve the bit error performance of the partial response CPM scheme, their effects on the bandwidth of the modulated signal must be observed. In increasing L, it has been shown that the bandwidth is reduced. However, increasing M is likely to increase bandwidth also. Thus, while increases in L will result in an overall improvement in performance efficiency, the same does not apply for M. When M is varied, a trade off between bit error performance and bandwidth utilisation occurs. Therefore, it is important to select M carefully to obtain an optimum performance.

Decreasing h and using pulse shaping will further enhance the bandwidth efficiency of the CPM scheme in general. However, caution must be exercised when selecting h. As was shown earlier, weak modulation indices and small N_t may cause serious bit error degradation. Although increasing L and N_t do not adversely affect bandwidth efficiency, it will increase receiver memory requirements. Pulse shaping is also likely to increase hardware complexity.

While it is apparent that increasing L and N_t, together with pulse shaping can improve the overall performance efficiency of a CPM scheme, it is not obvious what performance trade off exists for M and h. This is because the overall performance efficiency of a modulation scheme is based on two main criteria, *bit error performance* and *bandwidth efficiency*. In the next section, a method of comparing both to determine the overall performance efficiency will be provided.

2.9 Comparison of Digital Modulation Schemes

The overall performance efficiency of a digital modulation scheme is determined by both its bit error performance and bandwidth efficiency [7].

The bit error performance of a modulation scheme can be defined as the probability of bit error for a given signal-to-noise (SNR) ratio of the modulated signal in an AWGN channel. The probability of bit error can be derived analytically or via simulation. For

complex modulation schemes, the theoretical analysis required can be very complicated, if not impossible.

Before more is said about bit error performance analysis, it is important to understand what is meant by AWGN. *Noise* is a term commonly used to designate unwanted waveforms that tend to disturb the transmission and processing of signals in communication systems. Although there are many sources of noise, they can fall into two main categories - internal and external to the system. External noise may be due to atmospheric noise, galactic noise and man-made noise while internal noise includes an important type of noise that is caused by spontaneous perturbations of current or voltage in electrical circuits. Whether it is in small or large intensities, this form of noise is present in every communication system and originates at the front end of the receiver part of a system. It is commonly referred to as *receiver noise* [7].

In modelling noise, an idealised form called *white noise* is customarily used to include both external and receiver noise. The power spectral density of this noise is independent of frequency. The term "white" is used in the sense that white light contains equal intensities of all frequencies within the visible band of electromagnetic radiation [7]. Since this noise is modelled as a Gaussian random variable and is normally added at the channel, it is more specifically termed *additive white Gaussian noise* (AWGN).

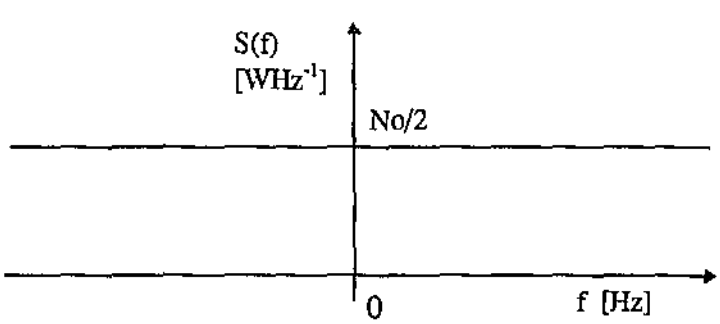

Figure 2.9-1: The power spectral density of AWGN.

A basic simulation model to obtain the bit error performance over an AWGN is shown in Figure 2.9-2 below. For a given signal s(t) and AWGN strength n(t), a random symbol sequence is generated by the symbol generator which is then modulated and passed through an AWGN channel. The received signal is then demodulated to obtain the

received data sequence $\hat{\mathbf{I}}$. Subsequently, a symbol comparator is used to compare the received data sequence $\hat{\mathbf{I}}$ against the original **I** from which a bit error probability is obtained. The bit error probability determined by the symbol comparator is obtained by dividing the total number of incorrect symbols received by the total number of symbols transmitted. Since the SNR is fixed for the duration of the analysis, the bit error probability only applies to that particular SNR. By repeating this procedure for various SNRs, a plot of the bit error performance against SNR per bit can be obtained.

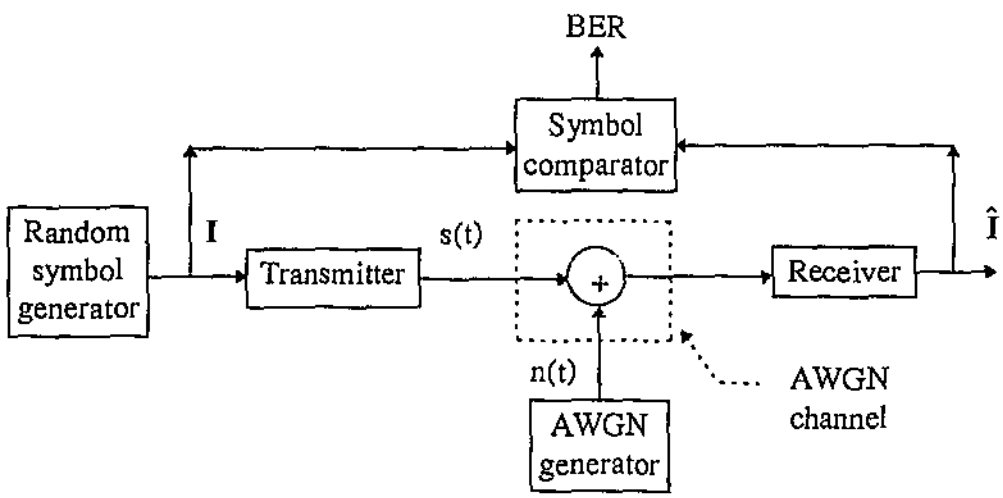

Figure 2.9-2: A basic simulation model to obtain the bit error performance.

In order to obtain a meaningful comparison, the bit error performance plots are obtained by plotting the *probability of bit error* against *Eb/No*. The energy per bit Eb of the modulated signal is denoted as [7]

$$Eb = PT/n \qquad (2.9\text{-}1)$$

where P is the average power of the modulated signal, T is the symbol period, and n is the number of bits per symbol. No/2 is the power spectral density of AWGN and was illustrated by Figure 2.9-1. As an example, Figure 2.9-3 illustrates the bit error performance plots of different Phase Shift Keying (PSK) and Frequency Shift Keying (FSK) systems.

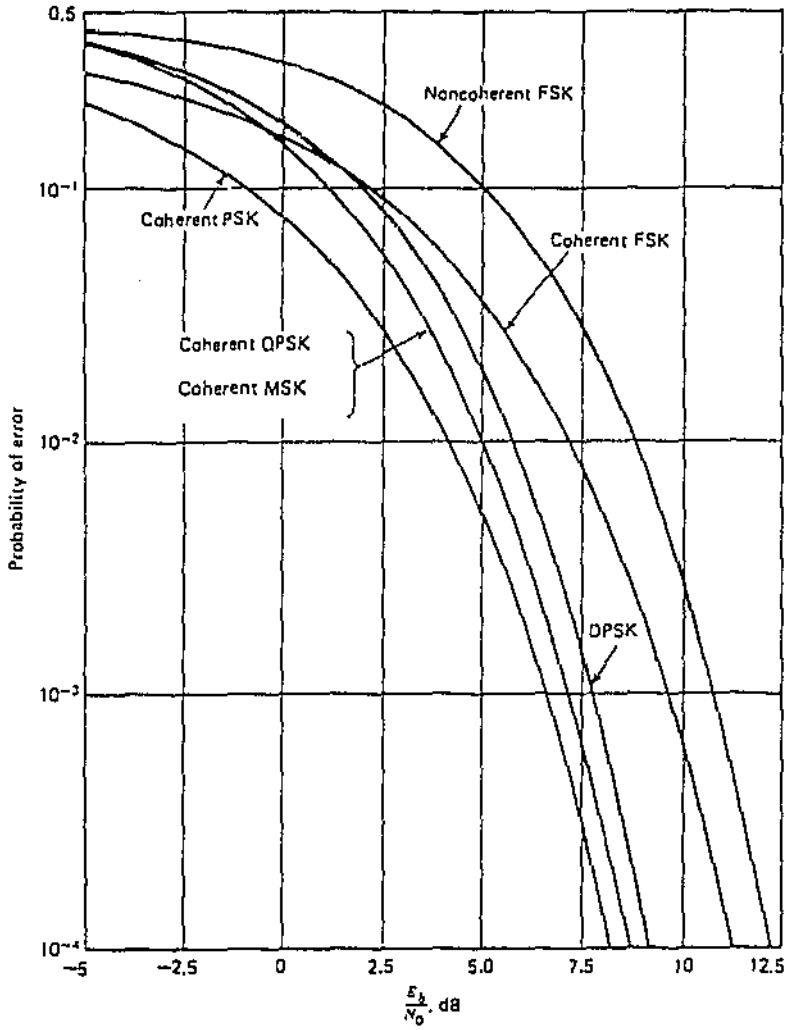

Figure 2.9-3: Bit error performance of different PSK and FSK systems [7].

For an accurate bit error performance analysis, the Monte Carlo [4] simulation technique can be employed to ensure that a lower bound on the accuracy is maintained. The Monte Carlo technique will be described in detail later.

The bit error performance plot is also used to compare different types of detectors. For different detectors, the bit error performance plots are likely to vary for the same modulation scheme. Hence, the type of detector used to obtain the bit error performance plot is usually specified clearly.

The bandwidth of a digital modulation scheme can be determined by mapping the modulated signal waveform into the frequency domain analytically using techniques such

as Fourier transform and autocorrelation function. However, this approach is normally very tedious for non linear modulations with memory such as CPM. It is far more convenient to obtain the results from simulation if the bit error performance is also to be obtained in the same way.

The *bandwidth efficiency* of a digital modulation scheme is measured by the bit rate to bandwidth ratio, which is [8]

$$\text{Bandwidth Efficiency} = B/Rb \qquad (2.9\text{-}2)$$

where B is the half bandwidth of the modulated signal and Rb is the bit rate of the data sequence [7].

A concise and appropriate comparison of digital modulation schemes is one based on the bandwidth efficiency versus Eb/No required to achieve a given error probability [8]. This method of comparison is normally performed on a graph like the one shown in Figure 2.9-4. For a desired error probability, the required *Eb/No* value is determined from the bit error performance plot. Together with the bandwidth efficiency, a point is plotted on a graph where it coincides with the two performance indicators on both axes. Figure 2.9-4 illustrates this method of comparison for M-ary PSK and M-ary FSK schemes.

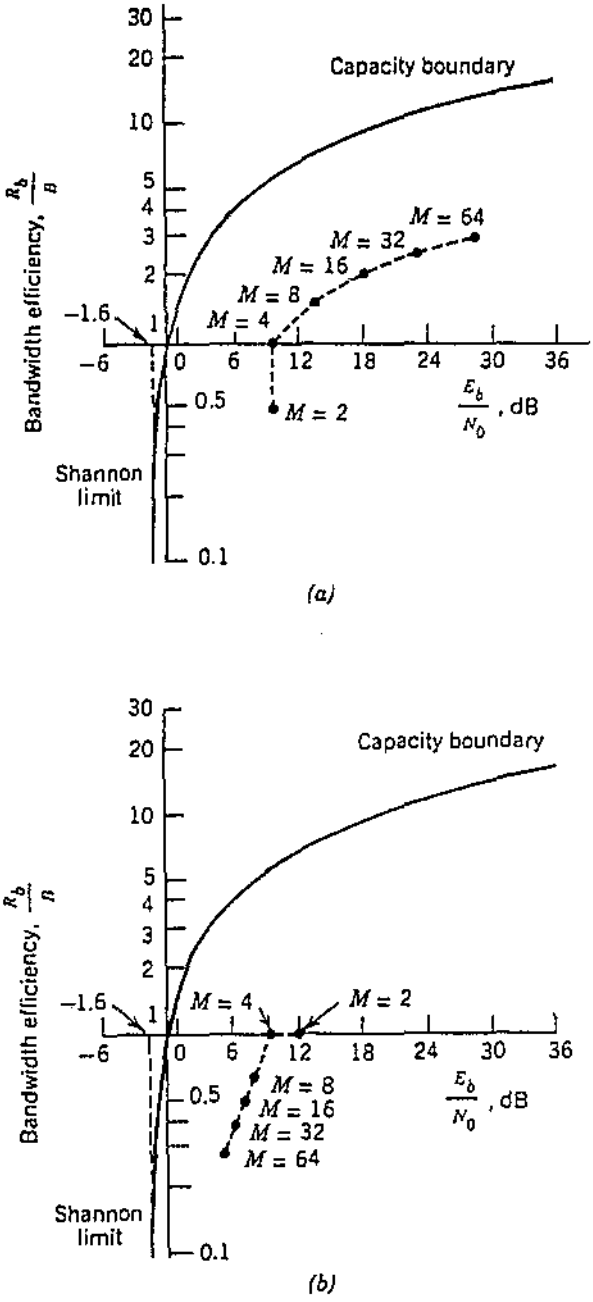

Figure 2.9-4: (a) A comparison of M-ary PSK with the ideal system. (b) A comparison of M-ary FSK with the ideal system [7].

It is obvious that for a given error probability, the most efficient modulation scheme is one with the highest bandwidth efficiency coupled with the lowest Eb/No required to maintain the desired bit error performance. However, to effectively facilitate an absolute comparison of modulation schemes, it is necessary to also plot the Shannon limit [7]. *The*

Shannon's channel capacity theorem states that in a band-limited communication channel that is disturbed by AWGN and is subjected to a power constraint, the channel capacity C (in bits per second) is defined by

$$C = B \log_2(1 + SNR) \tag{2.9-3}$$

where B is the channel's half bandwidth (in hertz), and SNR denotes the received signal-to-noise ratio [7]. The channel capacity C sets an upper limit on the rate at which information may be transmitted without error. Hence, it serves as the upper bound on the bandwidth efficiency of any type of modulation. The theorem can be expressed as

$$\frac{E_b}{N_0} \leq \frac{2^{\frac{R_b}{B}} - 1}{\frac{R_b}{B}} \tag{2.9-4}$$

where R_b is the data bit rate [7]. This expression suggests that for a specified bandwidth efficiency R_b/B, the received signal Eb/No must satisfy (2.9-4) if transmission over the channel is to be error-free. Figure 2.9-4 shows the capacity curve obtained by plotting (2.9-4) when it is satisfied with equality. The closer a point representing a modulation scheme is to the capacity boundary, the more efficient it is in the overall sense in terms of bandwidth efficiency and bit error performance [7].

3 Model of an Indoor Wireless Channel

This chapter explains the nature of an indoor microwave channel and proposes a composite Rician channel model for the purposes of simulation.

In many radio channels, there may be more than one propagation path from the transmitter to the receiver. These multipath propagations may be caused by atmospheric reflection or refraction, or reflections from buildings and other objects. They result in fluctuations in the received signal level. Multipath effects in an indoor microwave channel, as with any indoor radio channel, is a major problem. This is mainly due to the confined nature of the propagation space which is typically bounded by walls and obstructed by objects. Figure 3-1 illustrates multipath propagation in an indoor environment

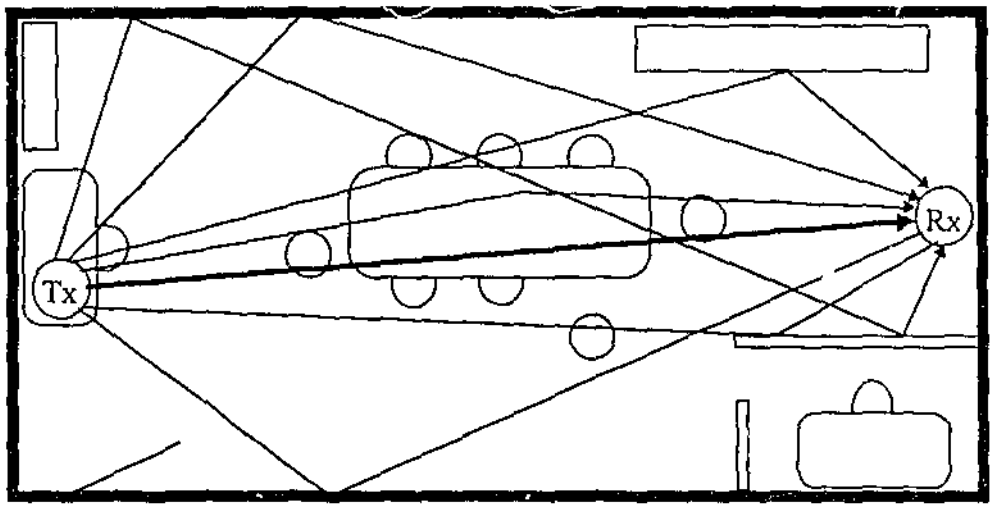

Figure 3-1: Multipath propagation in a typical indoor environment with furniture and cubicle partitions. The rays from the transmitter Tx to receiver Rx are the multipath signals with the bold ray signifying the direct line-of-sight propagation.

In modelling a real channel, it is also useful to consider the effects of the noise and distortions introduced by the transmitter and receiver. If a lookup table transmitter like the one shown in Figure 2.4-2 is used, it is unlikely that the transmitter will cause any significant adverse effects. In contrast, the receiver is likely to cause some problems due

to the filters used in the quadrature receiver. The quadrature receiver, as shown in Figure 2.7-1, is typically used in the reception of modulated radio signals. The non-ideal low pass filters required to cut out double frequency components can cause distortions due to their imperfect cut-offs and delays. In the composite channel models to be proposed later, the effects of quadrature reception is accounted as part of the model.

In the next section, a composite AWGN channel model is presented without considering the effects of multipath interference. This model is *composite* because it considers two important influences - AWGN and quadrature reception. Subsequently, in section 3.2 another composite channel model which accounts for AWGN, quadrature reception, as well as multipath interference is developed and the final model presented.

3.1 Composite AWGN Channel Model

In the model of the composite AWGN channel below, the AWGN is added to the modulated signal s(t) to form the received signal $\hat{r}(t)$ which is then received by the quadrature receiver to form the in-phase the quadrature components $\hat{I}(t)$ and $\hat{Q}(t)$.

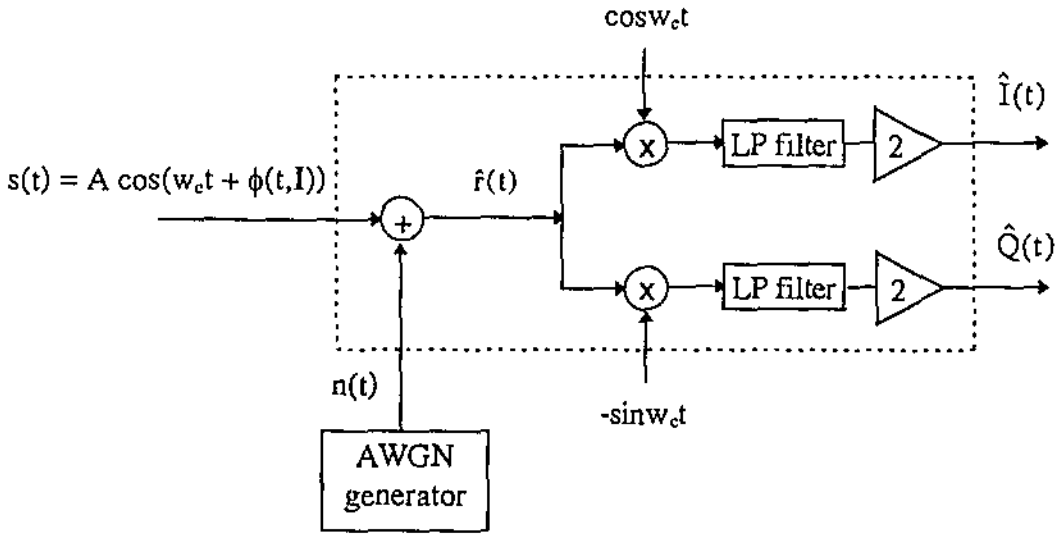

Figure 3.1-1: Composite AWGN channel.

3.2 Composite Rician Channel Model with AWGN

Reference [17] reported experiments to determine the statistics related to the random variation of modulated signals in indoor multipath environments. The measurements were measured in two types of buildings. It was originally anticipated that the distribution function of the received signal for fixed receiver locations with no line-of-sight path to the transmitter should correspond well with the Rayleigh distribution. Instead, the distribution functions computed from the data at all locations were found to be Rician. This finding has also been supported in other literatures reviewed for this project. Hence, a Rician multipath model was used for the composite channel model described in this section. The multipath model is also *fast-fading* and *frequency non-selective* which is appropriate for modelling motion within the physical multipath channel of a narrowband transmission [17] [9].

Figure 3.2-1 is a representation of a multipath channel. The various paths of propagation are the reflections of s(t) which have been attenuated by a factor of $\alpha_n(t)$ and delayed by $\tau_n(t)$ due to the structure of the medium. The factor $\alpha_n(t)$ is the attenuation factor for the signal received on the nth path and $\tau_n(t)$ is the propagation delay for the nth path. Since the attenuation and delay are expressed as functions of time, the reflected signals are time variant [4].

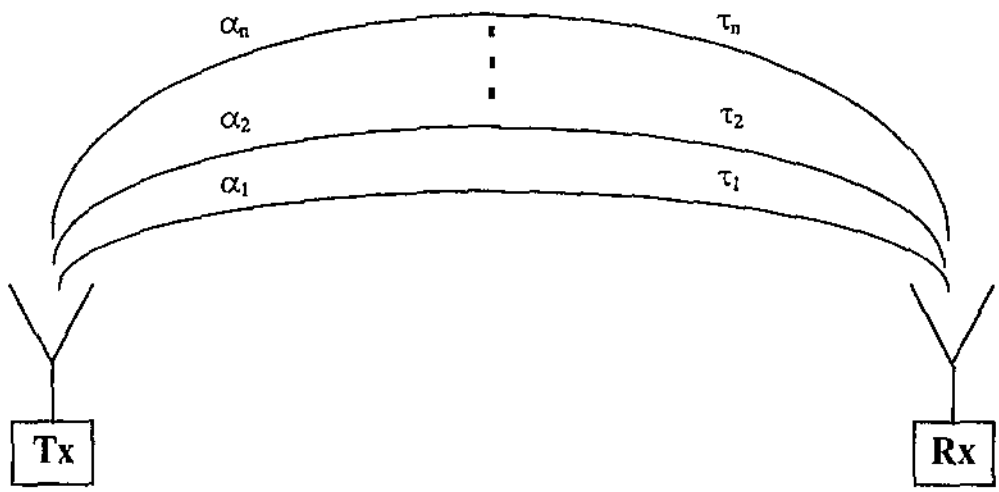

Figure 3.2-1: Simplified illustration to show the different path lengths of a multipath propagation [4].

Consider the effects of the multipath channel on a transmitted bandpass signal that is represented in general as [5]:

$$s(t) = \mathrm{Re}[s_l(t)e^{j2\pi f_c t}] \tag{3.2-1}$$

Assume that there are multiple propagation paths as shown in Figure 3.2-1. Thus, the received bandpass signal may be expressed in the form [8]:

$$x(t) = \sum_n \alpha_n(t)s(t - \tau_n(t)) \tag{3.2-2}$$

As an example, a simple seven ray model utilising six reflected paths and one unattenuated direct line-of-sight component is shown in the Figure 3.2-2. This model may be used to model the six reflections bouncing off the six bounding surfaces of a room. In reality, the number of interfering multipath propagations are much larger and the interference is much more complex. In the following section, a more realistic and comprehensive channel model will be presented.

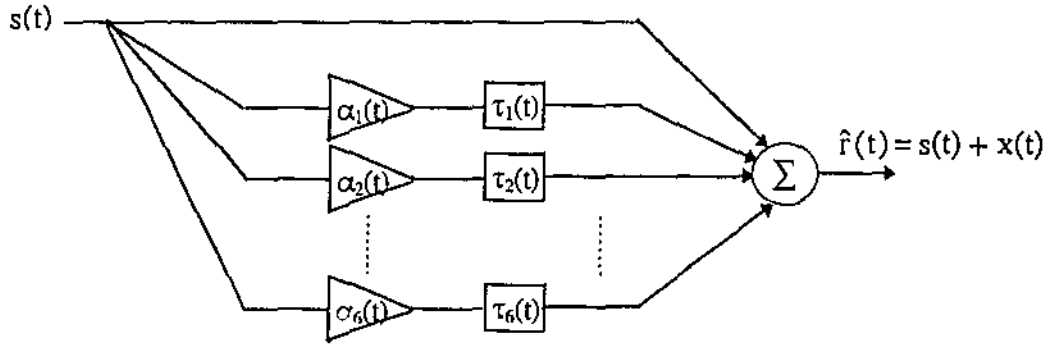

Figure 3.2-2: A simplified seven ray model.

By substituting (3.2-1) into (3.2-2), the following expression is obtained [8]:

$$x(t) = \mathrm{Re}\left\{\left[\sum_n \alpha_n(t)e^{-j2\pi f_c\tau_n(t)}s_l(t - \tau_n(t))\right]e^{j2\pi f_c t}\right\} \tag{3.2-3}$$

From (3.2-3) the equivalent low pass received signal is [8]:

$$r_l(t) = \sum_n \alpha_n(t)e^{-j2\pi f_c\tau_n(t)}s_l(t - \tau_n(t)) \tag{3.2-4}$$

Now, consider the transmission of an unmodulated carrier at frequency f_c. Thus, $s_l(t) = 1$ for all t, and consequently the received signal for the case of discrete multipath reduces to

$$r_l(t) = \sum_n \alpha_n(t) e^{-j2\pi f_c \tau_n(t)}$$

$$= \sum_n \alpha_n(t) e^{-j\lambda_n(t)} \qquad (3.2\text{-}5)$$

where $\lambda_n(t) = 2\pi f_c \tau_n(t)$ [8]. Hence, the received signal can be observed as the sum of a number of time-variant vectors (phasors) having amplitudes $\alpha_n(t)$ and phases $\lambda_n(t)$. For a large change in $\alpha_n(t)$ to occur such that a significant change is observed in the received signal, large dynamic changes in the medium needs to occur. In contrast, $\lambda_n(t)$ can change dramatically with relatively small motions of the medium. The delays $\tau_n(t)$ associated with the different signal paths can be expected to change at different rates and in an unpredictable manner [8]. Thus, the received signal $r_l(t)$ can be modelled as a random process. When the number of paths is large, the central limit theorem can be applied such that $r_l(t)$ may be modelled as a complex-valued Gaussian random process [8]. With that understanding and using (3.2-1), the unmodulated bandpass signal may be expressed as

$$s_{unmodulated}(t) = \mathrm{Re}\left[(\zeta + j\xi)\, e^{j2\pi f_c t}\right] \qquad (3.2\text{-}6)$$

where ζ and ξ are Gaussian variables [8]. This expression can be rewritten as:

$$s_{unmodulated}(t) = \zeta \cos w_c t + \xi \sin w_c t \qquad (3.2\text{-}7)$$

Expression (3.2-7) does not account for the direct line-of-sight component. By including the unmodulated direct line-of-sight component $A\cos(w_c t)$, the unmodulated carrier becomes

$$s_{unmodulated}(t) = (\zeta + A)\cos w_c t + \xi \sin w_c t \qquad (3.2\text{-}8)$$

where $A = \sqrt{\frac{2E}{T}}$ [8]. Expression (3.2-8) is shown by [9] to represent a Rician fading channel if ζ and ξ are independently Gaussian-distributed random variables with identical probability density functions of zero mean and variance equal to the local mean reflected power q_s. To show that the amplitude of $s_{unmodulated}(t)$ is a Rician distribution, expression (3.2-8) is represented as [9]

$$s_{unmodulated}(t) = \rho\cos(2\pi f_c t + \varphi) \tag{3.2-9}$$

with

$$\rho = \sqrt{(A+\zeta)^2 + \xi^2} \tag{3.2-10}$$

$$\varphi = \arctan\left(\frac{A+\zeta}{\xi}\right) \tag{3.2-11}$$

It is now evident that (3.2-10) is related to the non-central chi-square distribution. Hence, the instantaneous amplitude ρ has the Rician probability density function

$$f_\rho(\rho|q_s, A) = \frac{\rho}{q_s}\exp\left(-\frac{\rho^2 + A^2}{2q_s}\right) I_0\left(\frac{\rho A}{q_s}\right) \tag{3.2-12}$$

where $I_0(\bullet)$ is the modified Bessel function of the first kind of zero order [9]. The local mean power is provided as

$$p = q_d + q_s \tag{3.2-13}$$

where $q_d = \frac{1}{2}A^2$ is the power in the direct component and q_s is the average power of the scattered component [9]. The K factor is defined as the ratio of direct power and scattered local mean power.

$$K = \frac{q_d}{q_s} \tag{3.2-14}$$

The rate of fading in a channel is determined by rate at which the amplitude of the received carrier changes. If there is rapid motion of objects within the medium, it is likely to result in fast variations of the amplitude [8]. When this occurs, the channel is *fast fading* otherwise, if the rate of change is slower or equal to the symbol rate R_s, the channel is *slow fading* [8]. To model a slow fading multipath channel, the rate of change of the Gaussian random variables ζ and ξ may be set to the symbol rate R_s or below. To achieve fast fading, R_s should be increased beyond the symbol rate [8].

It is important to note that the expressions for the received carrier is unmodulated. It was possible to use the central limit theorem to approximate the scattered baseband components as a complex Gaussian distribution because the derivation considered them to be unmodulated. In the Rician channel model to be proposed, the carriers are

modulated. This model is presented on the assumption that the interdependency of the scattered baseband component due to the common information bearing phase signal is not significant enough to change the amplitude distribution of the received signal s(t).

The channel model illustrated in Figure 3.2-2 accounts for the multipath propagation of both scattered and direct line-of-sight components based on the Rician distribution discussed earlier. It also accounts for AWGN and the filtering effects of quadrature reception. The model is suitable for simulation and is implemented as part of a simulator in this thesis.

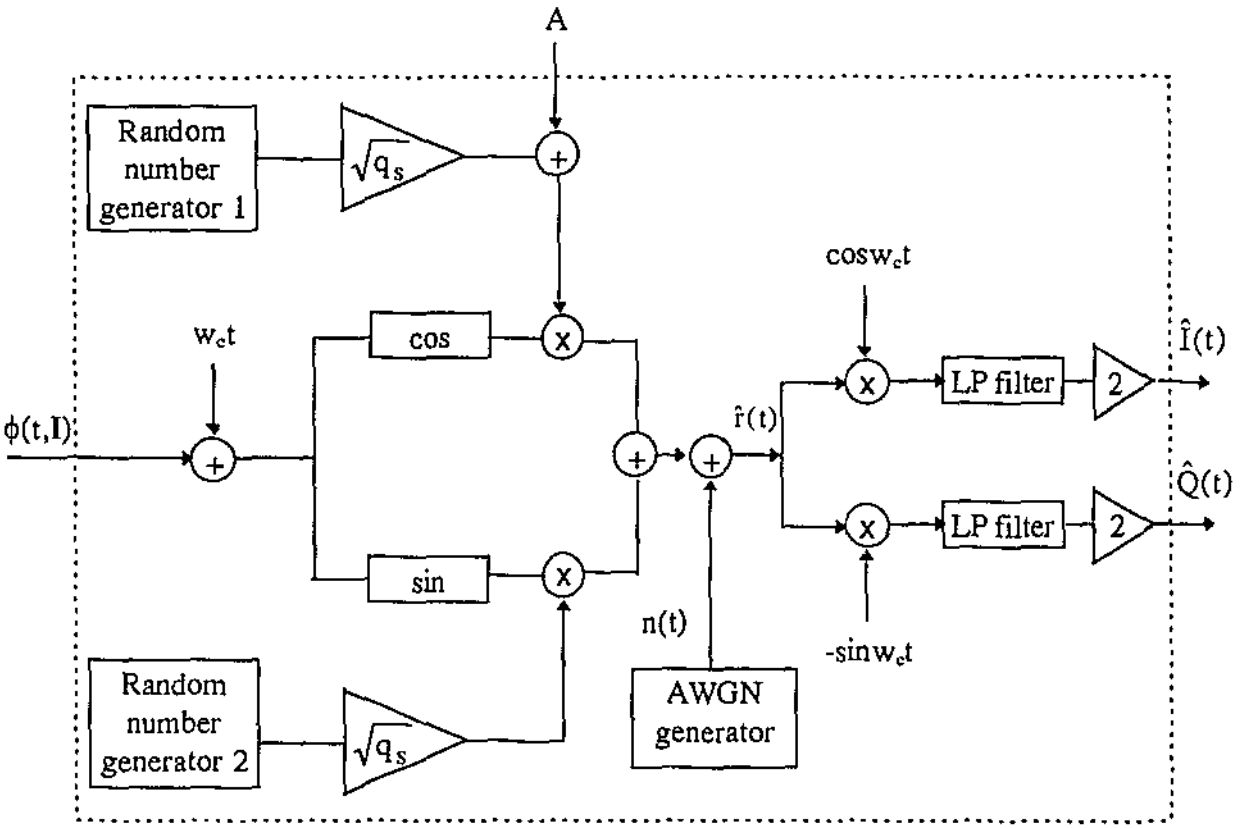

Figure 3.2-2: A composite Rician channel model with AWGN

4 SIMULINK Implementation of a CPM Simulator

Three simulators were constructed to perform three different tasks - to obtain the bandwidth occupancy of CPM signals, to determine the bit error rate performance of CPM schemes over a composite AWGN channel, and finally, to determine the bit error rate performance of CPM schemes over a Rician channel with AWGN. The SIMULINK schematic of these models are presented in figures 4-1 to 4-3. These simulators were developed entirely as digital systems. For convenience, these models are referred to as Simulator 1, Simulator 2 and Simulator 3 respectively. The MATLAB and SIMULINK source codes for these simulators are available in Appendices A and B.

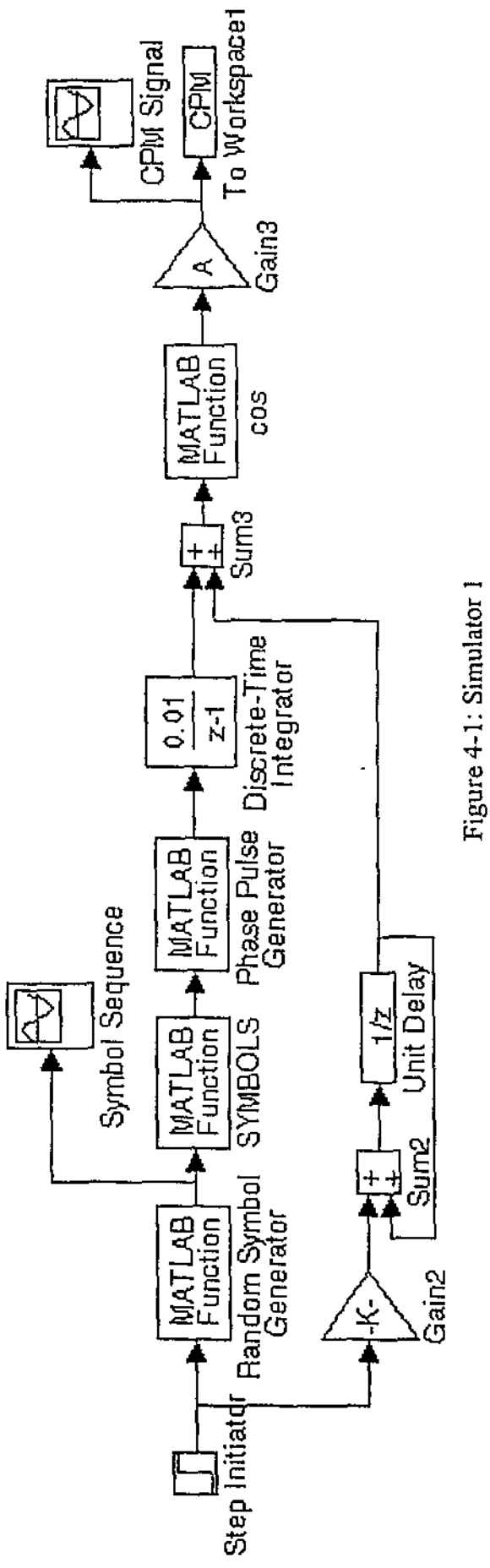

Figure 4-1: Simulator 1

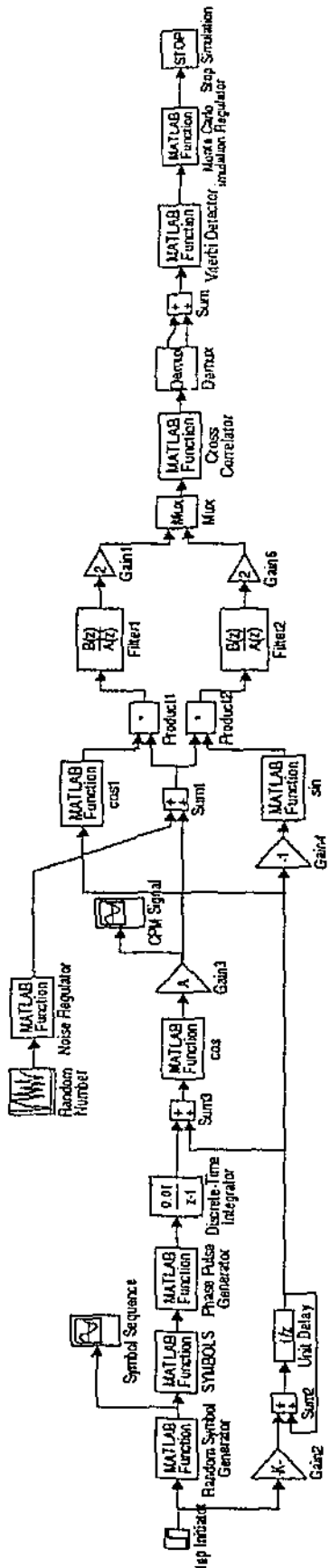

Figure 4-2: Simulator 2

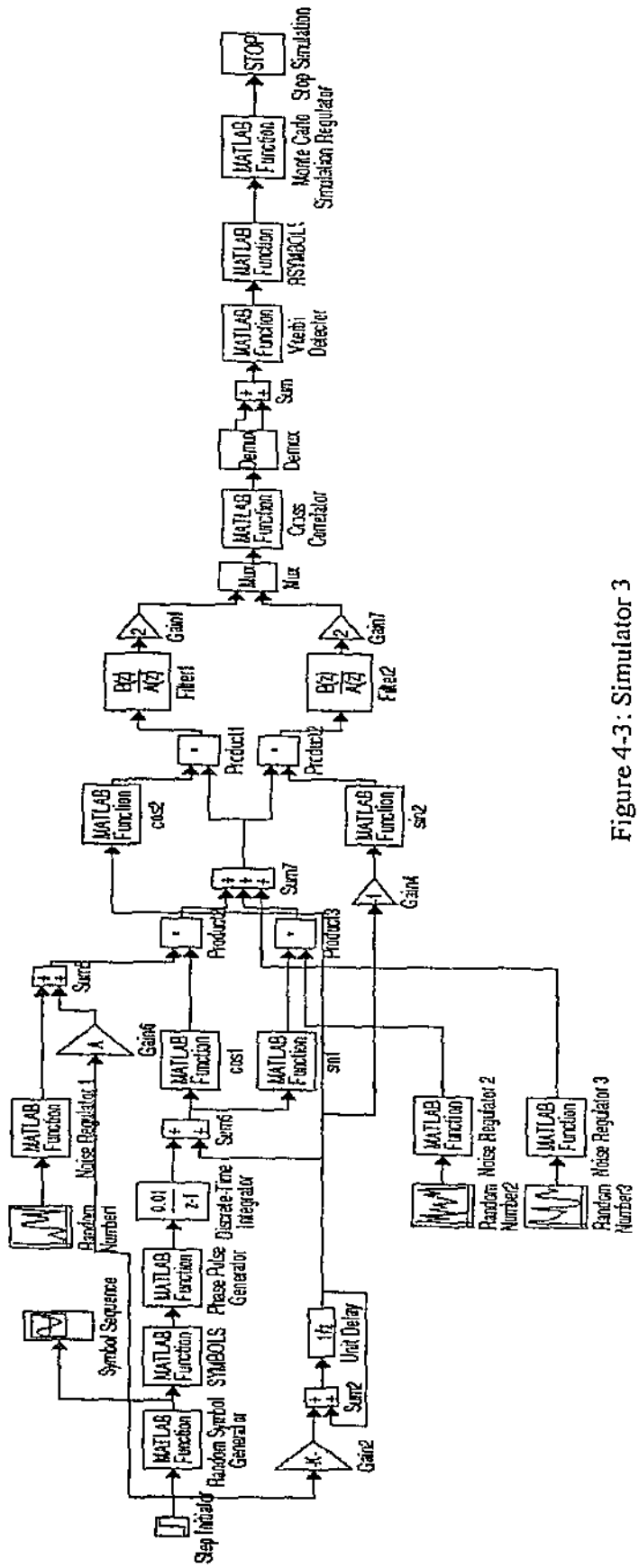

Figure 4-3: Simulator 3

4.1 Overview of SIMULINK

SIMULINK is a program for simulating dynamic systems. It is a visual extension of MATLAB with many additional features specific to dynamic systems while retaining all of MATLAB's general purpose functionality [11].

SIMULINK provides a visual interface from which a system can be designed either from pre-built or user-defined blocks. These blocks are interconnected by drawing signal lines and using special interconnection blocks to create a block diagram defining the desired system. Once completed, the simulation can be performed computationally while the user watches the results of the simulation via probes attached to strategic points on the signal line network. Alternatively, the results of the simulations may be viewed in MATLAB where special sink blocks have stored the simulation results into workspace variables. In MATLAB's command line workspace environment, the workspace variables may be viewed by simply typing the variable name or by plotting it to a graph. It is also possible to pause the simulation to observe the performance of the model and restart it again from the state which it was paused.

When a system is designed purely on pre-built blocks, the system designer selects the desired blocks from a template library. Contained in this library is a multitude of blocks grouped into categories. These blocks represent analogue circuits, digital circuits, hardware connections, sources, filters, logic functions, memory, graphs and more. The designer would then select the desired blocks, define any parameters necessary in the blocks and join them up.

There are many advantages of using SIMULINK, the major benefits are listed below:

Intuitivity

SIMULINK offers the intuitivity of a visual design. It is possible to model many less complex systems, without any programming. Representing the model as a block diagram, is much more natural than representing it with programming code. The block diagram is also more a realistic and familiar form of representation for

system engineers. This enables designers to better understand the system behaviour.

Convenience

SIMULINK offers the convenience of pre-built blocks and MATLAB function libraries. For system designers, their precious time is best spent on actually designing the system rather than modelling components and constructing analysis tools. For most electronic systems, the basic building blocks and analysis tools are already provided in SIMULINK and MATLAB toolbox libraries. SIMULINK also provides the platform for which systems can be simulated. This means that the designer is not required to write the code to perform the simulation; he or she is only involved with modelling of the system.

Flexibility

It is very easy to monitor the progress of a simulation in SIMULINK. There are many types of probes which can be used to observe the signal flow through the lines connecting the blocks. Otherwise, the simulation can be paused so that system variables may be analysed, and restarted again from where it left off.

Modularity

The approach to system design in SIMULINK is essentially object-oriented. The objects (the blocks) are very much separate entities in that they communicate by sending signals to one another. This compares with procedural language where procedures communicate via shared data structures which is prone to unexpected changes. The high cohesiveness of the blocks and low coupling between them makes it easy to add, remove and modify the system.

Power of procedural language

When modelling large systems where some non linear blocks are very big and complex, it may be best to use MATLAB function blocks together with other typical SIMULINK blocks. In doing so, the strength of both SIMULINK, and MATLAB's procedural language can be exploited. It is important to mention that MATLAB's programming language is very productive for implementing vector processing and also very convenient because variables and data structures do not have to be declared and initialised.

4.2 Novel Approach for Amalgamating Elaborate MATLAB functions with SIMULINK

SIMULINK is capable of modelling linear and non linear, analogue and digital systems. Typically, systems are modelled by using either differential or difference equations. User defined blocks may be designed by state space equations. If the blocks are linear, the pre-built state space function block can be used. Otherwise, if the block is non linear, it may be defined using a text editor as S-Functions are stored away. State space function blocks can be used directly but S-Functions need to use pre-built S-Function blocks.

Another approach to defining non linear blocks is to define them as MATLAB functions. A pre-built block called the MATLAB function block is used to accommodate MATLAB functions in SIMULINK. This is the most general approach to constructing user-defined blocks which allows the system designer to create comparatively large and complex blocks. Unfortunately, the use of the MATLAB function block is typically limited to simple MATLAB functions. Most user defined blocks in MATLAB are created using S-Functions which uses state space equation techniques.

The approach taken to develop the simulators presented in this thesis was to exploit the power of MATLAB function blocks. This is an unconventional approach to designing SIMULINK systems which are typically modelled using state space techniques. The idea was to use MATLAB function blocks to create comparatively large and complex blocks and use pre-built blocks to join and support them. This technique is very convenient for modelling complex electronic systems such as the simulators presented in this thesis because the complexity of such systems can be accommodated by MATLAB functions while standard electronic components and connections can be constructed easily by integrating already pre-built blocks. Essentially, the technique proposed in this thesis culminates the major benefits of SIMULINK with the power of MATLAB's procedural language.

To effectively amalgamate large MATLAB functions into SIMULINK is not trivial. It should be understood that data processing in procedural language is typically performed

to the entirety of the data structure. For example, if a data structure was to be processed from one form to another, it would normally require a few steps where each step involves transforming the entire data structure to an intermediate form. In contrast, the data processing (signal processing) in SIMULINK involves transforming each segment of data (signal) from one form into another before the next data segment is processed. This type of simulation is common when simulating systems defined by differential or difference equations.

Thus, to adapt to this type of processing, the large MATLAB functions for the developed simulators of this thesis, process each signal sample at a time. These samples are then stored as a localised workspace variable until its length becomes large enough to facilitate the type of processing commonly associated with procedural languages. To keep track of the workspace variable length until it is long enough to be processed, a localised counter is employed. This special technique of using the workspace variable and the localised counter is unconventional but it is an effective and necessary technique to amalgamate large, complex MATLAB function blocks with SIMULINK.

4.3 Parameter File

Each simulator has a parameter file which serves two purposes. Firstly, it maintains a record of simulator parameters which are referenced by many of major blocks during simulation. Prior to simulation, the user would edit the parameter file to define the parameters of the CPM scheme, the characteristic of the channel, the behaviour of the receiver and regulation of the simulation. The file's second purpose is to generate the many lookup tables which will be required by the blocks in the progress of the simulation. Once, the user has entered the necessary parameters into the file, it is then executed from the MATLAB's command line. In executing the file, the lookup tables are generated and together with the parameters, they are lodged in memory as MATLAB's workspace variables. It is now possible for the blocks in SIMULINK to access these information. One of the major benefits of the parameter file is that it offers a localised area where modifications to the nature of the simulator can be conveniently adjusted

without tampering with the actual SIMULINK simulator. The design strategy was to ensure that the parameter file could span the entire range of the simulator's capability. In essence, this means that the parameter file is the "command centre" of the simulator, from which any type of simulation (within the simulator's scope) can be regulated. The parameter file serves another important function. It reduces simulation time by trying to pre-compute as much data as possible prior to simulation to reduce the work load of the simulator.

The parameter files for Simulators 1 to 3 are: inisimbw.m, iniac.m and inirc.m respectively. These files are included in a separate volume of documentation which contains all the source codes for the simulators.

4.4 Transmitter Implementation

The three simulators have the same transmitter. The transmitter was constructed from two major blocks - the random symbol generator and the phase waveform generator. These blocks are shown in Figure 4.4 which illustrates the SIMULINK schematic for the transmitter. The transmitter was developed as a direct mathematical implementation of expressions (2.2-1) and (2.1-3).

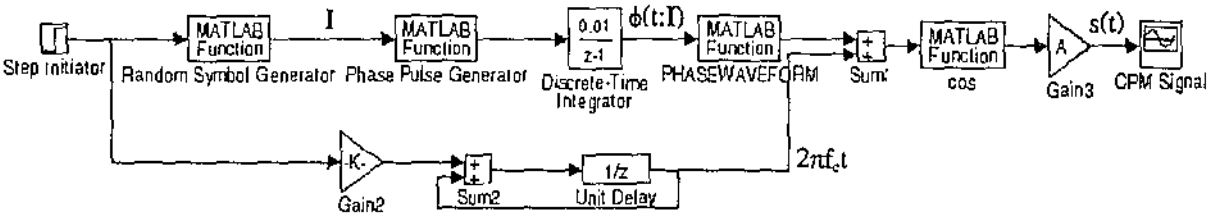

Figure 4.4-1: SIMULINK schematic of the transmitter with the equivalent continuous-time variables shown.

The random symbol generator generates M-ary data symbols according to the symbol rate specified in the parameter file. Depending on the modulation levels M, also specified in the parameter file, the generator will generate symbols randomly from the set {±1, ±3, ..., ± (M-1) }. To perform this, MATLAB's RAND( ) function was used to generate a

uniformly distributed random variable in the real number range of [0,1]. This range is then scaled to the total number of terms in symbol set. The randomly generated sequence is then discretised into integers using MATLAB's CEIL( ) function from which a mapping to the position of the symbol in the symbol set is used to generate the symbols randomly.

The frequency pulse shapes g(t) are pre-generated by the parameter file as part of an initialisation process before the simulation begins. During simulation, the phase pulse generator superimposes the predefined pulse shapes onto one another according to the derivative of expression (2.2-1). The output from the phase pulse generator is then integrated with a discrete time integrator to generate the information bearing phase waveform $\phi(t; \mathbf{I})$ which is subsequently modulated into the CPM signal.

In the phase pulse generator, the frequency pulse shapes are first multiplied with $2\pi I_k h_k$ and then superimposed onto a global variable which stores other pulses from previous symbol intervals. The global variable sample corresponding to the current time interval is then output. Superimposing is necessary as part of the implementation to generate partial response CPM schemes because the length of pulse shapes L are longer than one symbol period.

A trial of the transmitter was conducted for eight level CPM utilising raised cosine pulse shaping with modulation index h = 4 / 5 and pulse shape length 4 with the following simulation parameters:

- simulator sampling rate = 100Hz
- data symbol rate = 5Hz
- normalised intermediate carrier frequency = 40Hz

The results of the simulations are shown in figures 4.4-2 to 4.4-6. The horizontal axes for all five plots show time in units of sampling period ps = 0.01s. The first figure illustrates the raised cosine pulse g(t) which was pre-generated by the parameter file during the initialisation stage. Figure 4.4-3 shows the octanary data symbol sequence I generated by the random symbol generator while Figure 4.4-4 shows the superposition of

frequency pulses $2\pi \sum_{k=-\infty}^{n} I_k g(t-kT)$. Once integrated, the superposition of frequency pulses form the phase waveform $\phi(t, I)$ which is illustrated in Figure 4.4-5. Finally, Figure 4.4-6 illustrates the modulated CPM signal.

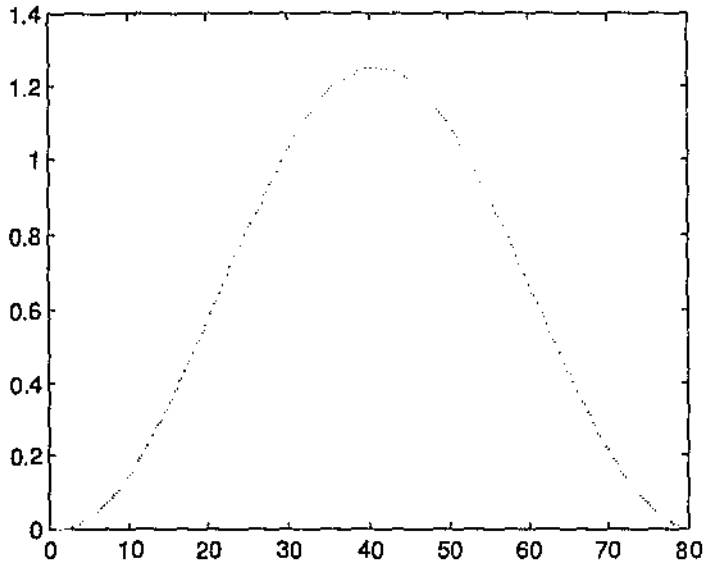

Figure 4.4-2: The pre-generated, raised cosine frequency pulse.

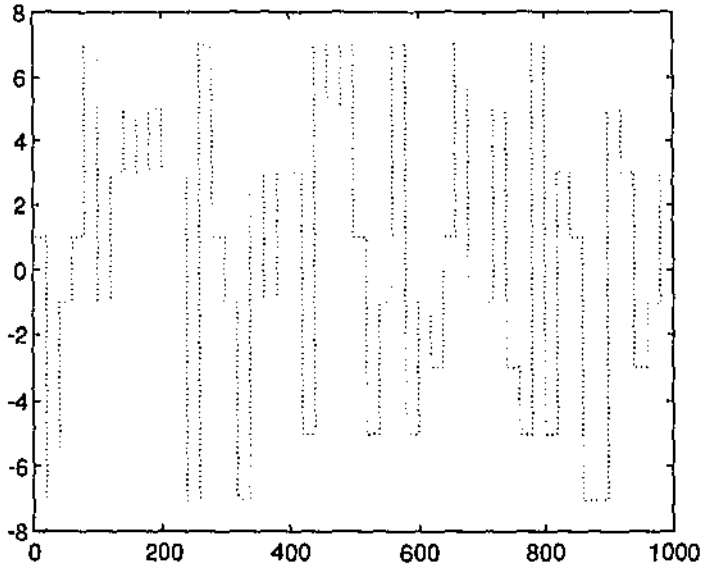

Figure 4.4-3: Random 8-level data sequence.

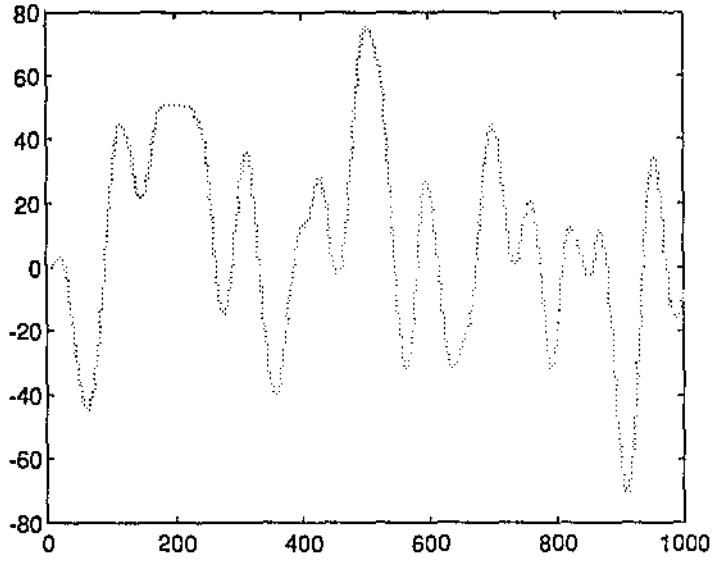

Figure 4.4-4: Superposition of frequency pulses.

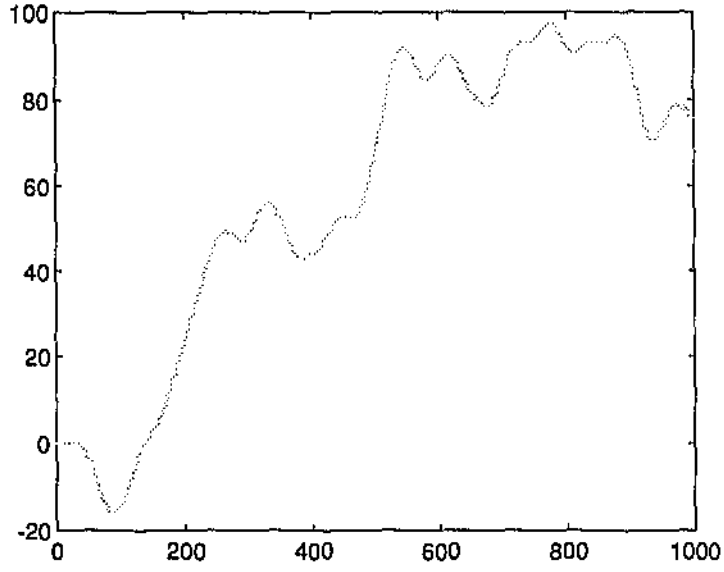

Figure 4.4-5: Phase waveform $\phi(t, I)$.

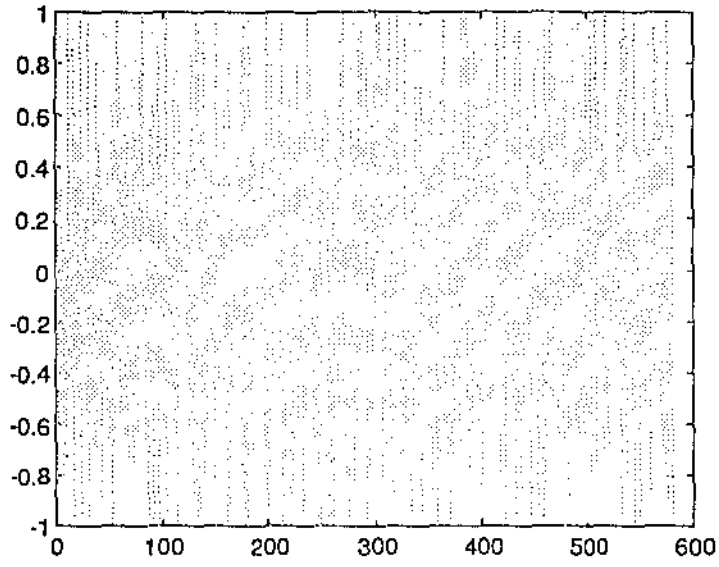

Figure 4.4-6: CPM waveform s(t,I).

4.5 Receiver Implementation

The implementation of the receiver in SIMULINK is shown Figure 4.5-1. The main components of the receiver are the cross correlator and the Viterbi detector. The inputs into the receiver are the in phase $\hat{I}(t)$ and quadrature $\hat{Q}(t)$ components of the received signal while the output is the detected data symbols $\hat{I}$. These signals were obtained from the quadrature receiver of section 3 where the quadrature receiver actually forms a part of the composite channel.

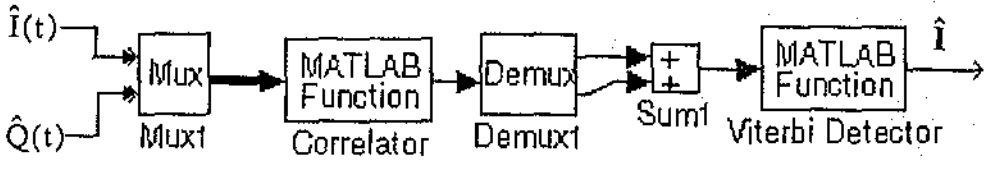

Figure 4.5-1: SIMULINK schematic of the receiver.

Cross correlator

For each symbol interval, the cross correlator reconstructs the received bandpass signal from the in phase and quadrature components. Once this is done, it cross correlates the bandpass waveform with all the possible waveforms corresponding to each state transition of the state trellis. These possible waveforms were pre-generated during the initialisation stage.

It is necessary to reconstruct the bandpass signal from the in-phase and quadrature components because then only one signal needs to be cross correlated with the possible waveforms matrix. Otherwise, two correlations need to be performed, one for the in-phase component and the other for the quadrature component. Since these correlations are done with the matrix of possible signal waveforms and at every symbol interval, it was decided to minimise the computational complexity by using only one correlation. It is important to note that the correlation is essentially baseband since the bandpass waveform is normalised to the fixed time interval t = [0,T].

The correlations performed between the received signal and the possible waveforms matrix for each symbol interval will provide an array of correlation metrics. Once computed, these metrics are then stored into a global variable table which will be subsequently used by the Viterbi detector.

The reconstructed bandpass signal for each symbol interval can be expressed as

$$\begin{aligned}\hat{s}_n(t) &= \hat{I}_n(t)\cos 2\pi f_c t - \hat{Q}_n(t)\sin 2\pi f_c t \\ &= \hat{A}\cos(2\pi f_c t + \hat{\phi}(t, I_k))\end{aligned} \tag{4.5-1}$$

where t is the range [0, T]. This waveform is then cross correlated with the possible waveforms matrix expressed as

$$W = \begin{bmatrix} w_1(t) \\ \vdots \\ w_a(t) \end{bmatrix} = \begin{bmatrix} w_{11} & \cdots & w_{1b} \\ \vdots & \ddots & \vdots \\ w_{a1} & \cdots & w_{ab} \end{bmatrix} \tag{4.5-2}$$

The number of samples per symbol interval is $b = T/p_s$, where T is the symbol period and p_s is the sampling interval of the simulator. The number of possible waveforms is

$a = pM^L$ (or $2pM^L$ if m is odd) for partial response schemes and pM (or 2pM) for full response schemes.

The cross correlation is actually done by MATLAB's vectorisable normalised cross correlation function CORRCOEF( ). The normalised cross correlation function can be expressed as

$$r = \frac{\int_T s_1(t)s_2(t)dt}{\sqrt{\int_T s_1{}^2(t)dt \int_T s_2{}^2(t)dt}} \qquad (4.5\text{-}3)$$

where r is a real number in the range [-1, 1].

Once the correlation is completed, CORRCOEF( ) returns a vector of size ($a \times 1$). The top most term, is the cross correlation of $w_1(t)$ with $\hat{s}_n(t)$, and the bottom most term is the cross correlation of $w_a(t)$ with $\hat{s}_n(t)$. With successive symbol intervals, each slice generated is stored in a MATLAB global variable called CCTAB and a timing signal is sent to the Viterbi detector. The CMs are transferred to the Viterbi detector in this way because otherwise it would be necessary to implement a vectored signal path between the correlator and the Viterbi detector. This would not be a generic implementation because depending on the CPM schemes being simulated, the size of the CCTAB will change while the size of the vectored signal path is fixed unless the SIMULINK schematic is physically modified. The storage of CM slices into CCTAB is illustrated in the following figure.

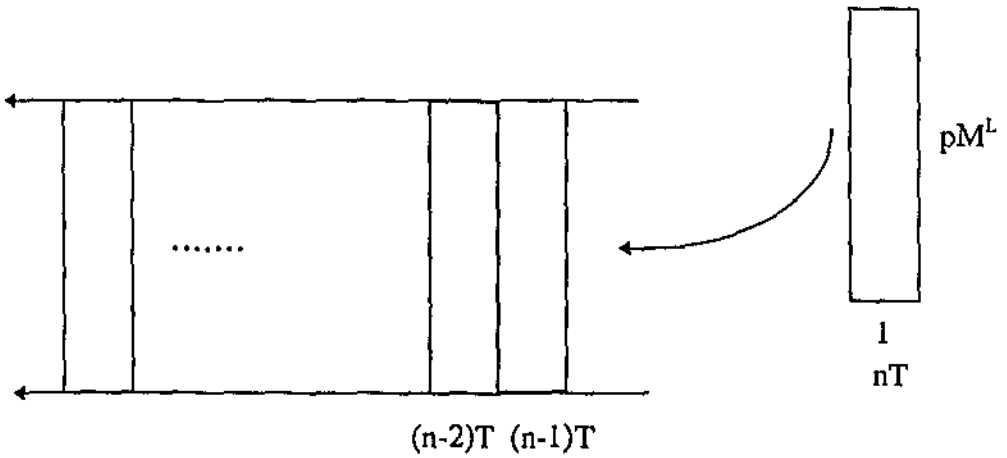

Figure 4.5-2: Storage of cross correlation metrics into CCTAB.

The generation of the possible waveforms matrix W involves generating the phase waveforms for all the possible state transitions. For both full and partial response schemes, the state transitions from state

$$S_n = \{\theta_n, I_{n-1}, I_{n-2}, \ldots, I_{n-L+1}\} \qquad (2.6.1\text{-}4)$$

to state

$$S_{n+1} = (\theta_{n+1}, I_n, I_{n+1}, \ldots, I_{n-L+2}) \qquad (2.6.1\text{-}6)$$

can be represented as:

$$ST = (\theta_n, I_n, I_{n+1}, \ldots, I_{n-L+2}) \qquad (4.5\text{-}4)$$

To generate all the possible phase waveforms due to the transition from any state to another, all the possible combinations of state transitions ST must be first generated. This list will provide all the possible combinations of initial phase state θ_n and L number of I's. For future reference, this list will be called the STLIST. The STLIST is then used to generate the possible phase waveforms by using expression (2.6.1-1) which is restated as

$$\phi(t;I) = 2\pi h \sum_{k=-\infty}^{n} I_k q(t-kT)$$

$$= \theta_n + 2\pi h \sum_{k=n-L+1}^{n} I_k q(t-kT), \qquad nT \le t \le (n+1)T \qquad (2.6.1\text{-}1)$$

Finally, the possible bandpass waveform matrix W is generated from the possible phase waveform matrix using the following expression where t is fixed in the range [0, T].

$$s(t) = \sqrt{\frac{2E}{T}} \cos\left[2\pi f_c t + \phi(t;I) + \phi_0\right] \qquad (2.1\text{-}3)$$

Viterbi detector

To implement the Viterbi detection technique, it is first necessary to obtain a lookup table which shows all the possible state transitions. This lookup table is generated by using the expression

$$\theta_{n+1} = \theta_n + \pi h I_{n-L+1} \qquad (2.6.1\text{-}7)$$

to perform an exhaustive search of the possibilities. In the actual implementation, the MATLAB variable which stores the table is called STATETTAB. STATETTAB is two

elements wide while the length is as long as the number of possible state transitions in the state trellis.

Since the Viterbi algorithm pertaining to the sliding window Viterbi detector has already been described, this section will be purely concentrated on its implementation.

The sliding window of the Viterbi detector always spans the last Nt slices of the CCTAB matrix. With a new symbol interval, the SWPs are extended by concatenating reachable states to the most recent ends of the SWPs using STATETTAB. For every extension, the corresponding metric from the cross correlator is added to the variable holding the cumulative CMs for that extended path. For every new state, the new SWP will be chosen from the extended paths such that it has the largest cumulative CMs of any other path arriving at that state. These new paths will then be stored in SPTAB and the corresponding cumulative CMs will be stored in CCSPTAB. The MLWP will then be chosen from SPTAB such that it has the largest value in CCSPTAB. The symbol detected will be chosen from the last state transition of the MLWP. The detected symbol will be selected such that it could have generated the transition. This is done by performing a backward search on STATETTAB.

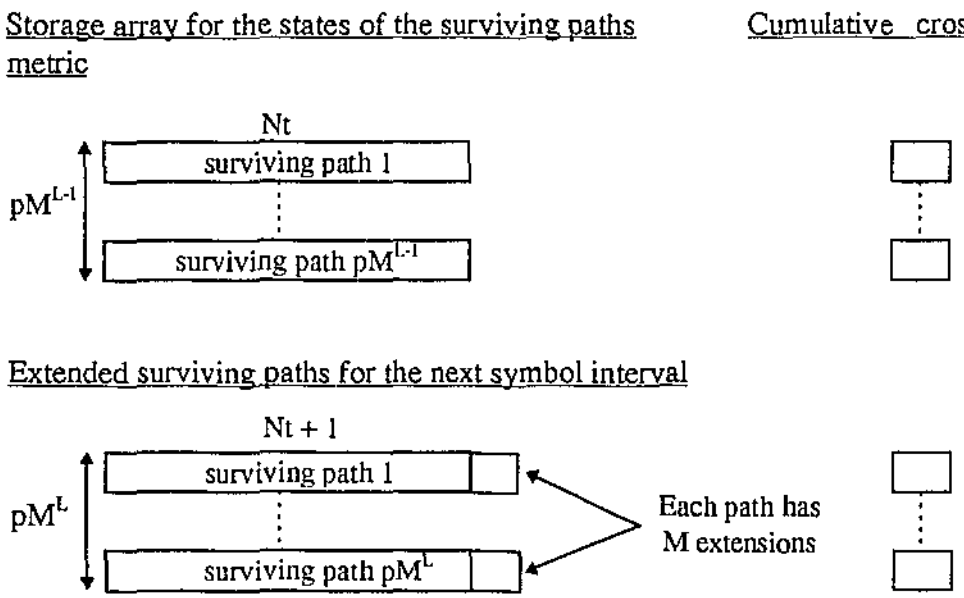

Figure 4.5-3: The main data structures for the Viterbi detector.

In the partial response scheme, this symbol is always unique because of the one-to-one mapping between state transition and symbol. However, with full response schemes this is not the case. There are actually pM (or 2pM) possible transitions (forward mapping) but only pp (4pp) possible state transitions (backward mapping). If p = 4 and M = 8, then the backward mapping is indeterministic. Consequently, the backward STATETTAB search technique is not useful for generic CPM. It is also computationally expensive to perform a backward search during simulation unless a backward STATETTAB was pre-generated at initialisation. To solve this problem, a list of symbols causing the forward state transition is generated together with STATETTAB. Thus when the SWPs are extended, the corresponding symbols from the list are recorded. That way there is no confusion with the symbols because every SWPs are recorded as both state and symbol transition paths.

Figures 4.5-4 to 4.5-7 were generated on Simulator 2 with the following simulation parameters:

- modulation levels M = 4
- frequency pulse length L = 2
- modulation index h = 0.5
- frequency pulse shaping type = LREC
- Viterbi detector sliding window size Nt = 10
- amplitude of modulated CPM signal A = 1
- normalised intermediate frequency carrier fc = 10 Hz
- SIMULINK's sampling frequency = 50 Hz
- data symbol rate = 4 Hz

Figures 4.5-5 to 4.5-7 were generated with AWGN of 5mW/Hz.

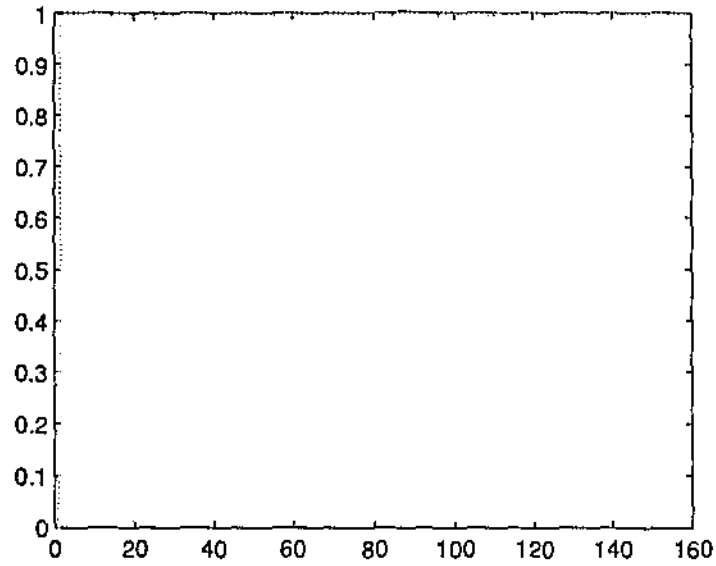

Figure 4.5-4: Near-perfect cross correlation metrics of the most likely path of a CPM scheme through a noiseless channel.

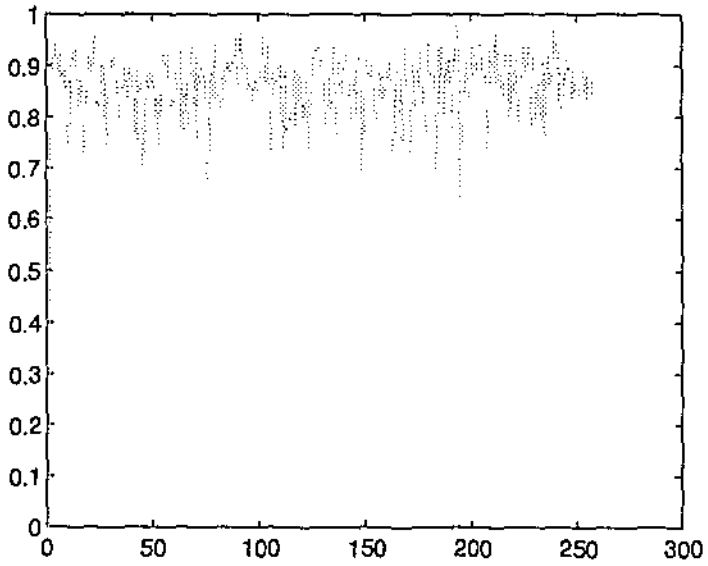

Figure 4.5-5: The cross correlation metrics of the most likely path of a CPM scheme through a composite AWGN channel with additive noise power spectral density of 5 mW/Hz. Detected symbol sequence had no errors.

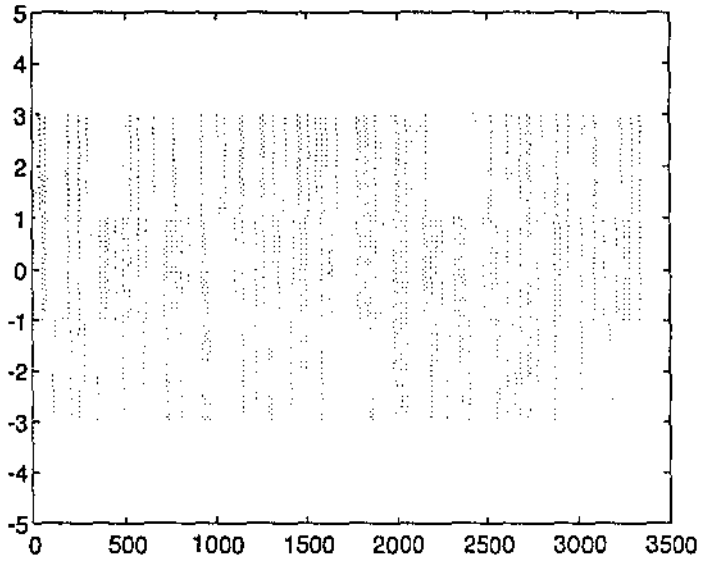

Figure 4.5-6: The transmitted symbol sequence.

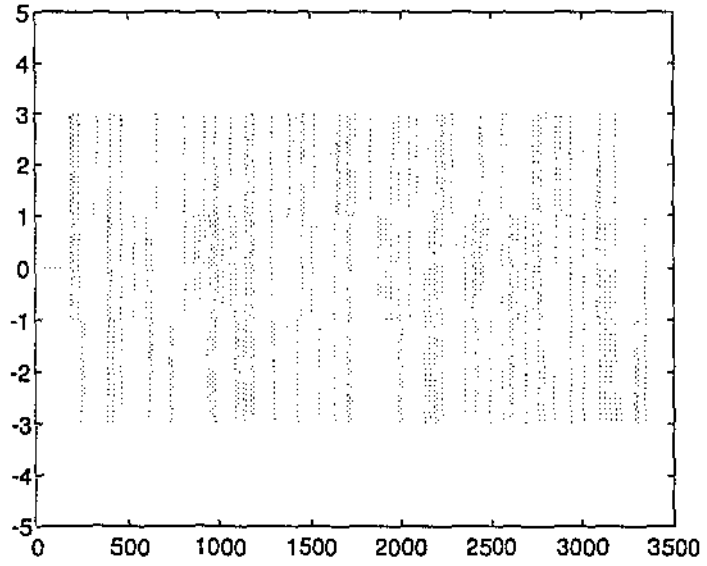

Figure 4.5-7: The received symbol sequence. No errors. Notice the initial delay.

4.6 Channel Implementation

The implementations of the channel models in SIMULINK for Simulator 2 and Simulator 3 are directly based on the composite channel models in sections 3.1 and 3.2 respectively. Since the purpose of Simulator 1 is to obtain the bandwidth of the modulated signal, a channel was not required for it.

The SIMULINK schematics for the channels of Simulator 2 and Simulator 3 are presented in Figure 4.6-1 and Figure 4.6-2 respectively. These models are digital as can be seen by the unit delay box and the digital filter notations. However, the equivalent continuous time signals are shown to help the reader understand the diagram.

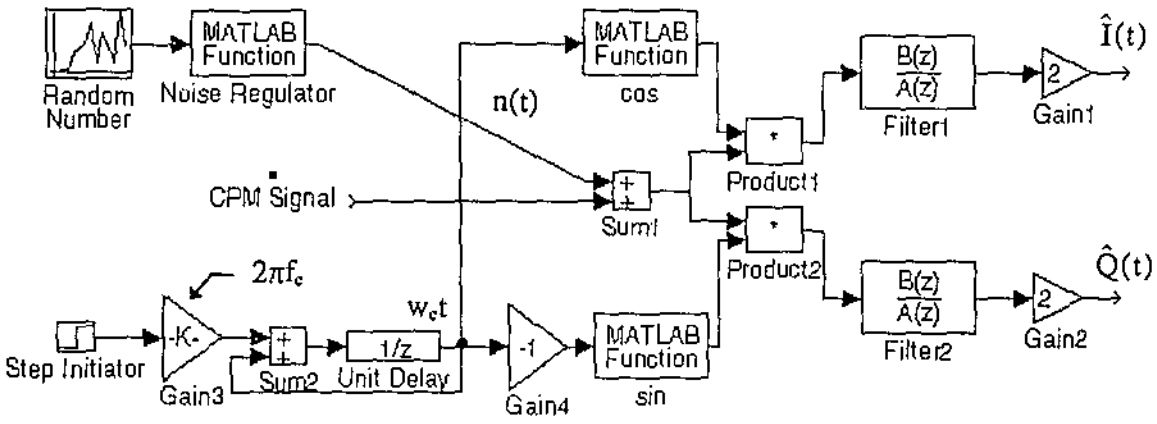

Figure 4.6-1: The implemented composite AWGN channel of Simulator 2.

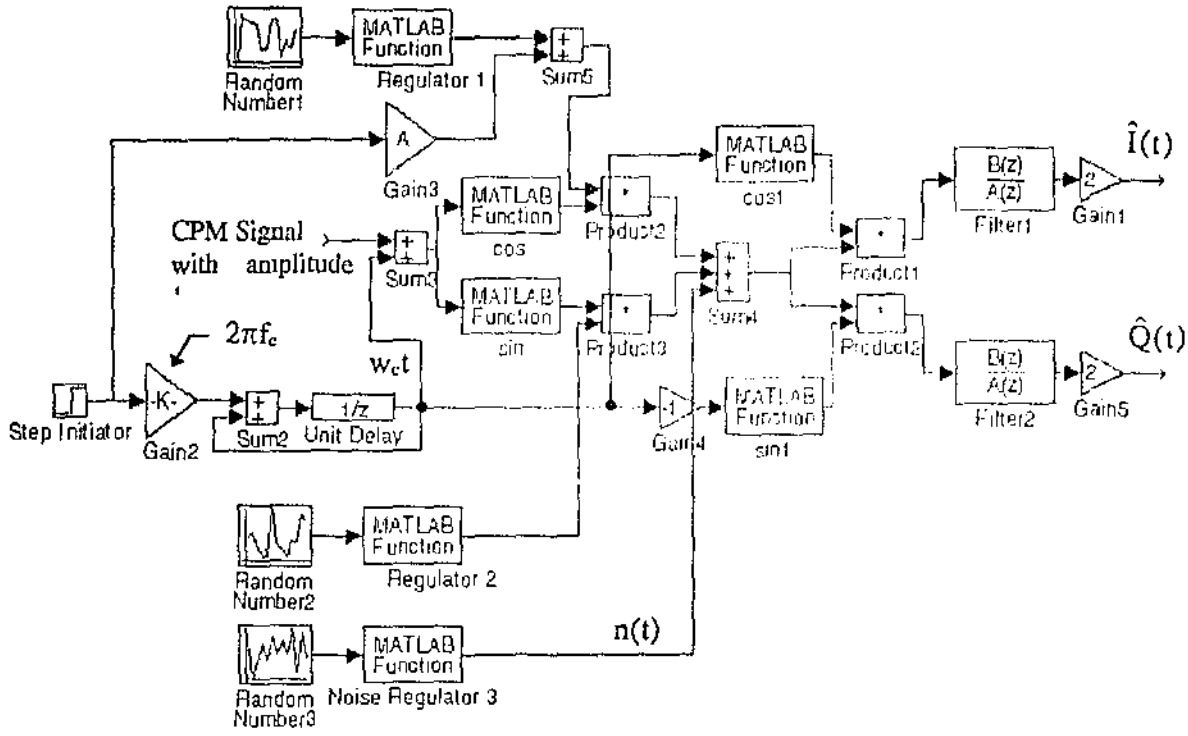

Figure 4.6-2: The implemented composite Rician channel with AWGN noise of Simulator 3.

To generate the AWGN in Simulator 2 and 3, a random number generator with a noise regulator is used. The random number block generates a random number at every sampling interval so it is equivalent to an AWGN generator which is bandlimited to half the sampling frequency $f_s/2$. The generated distribution has a mean of zero and a variance of one. Since the power of a random variable is also equal to its variance, the power of the AWGN is 1W. The power spectral density of the generated AWGN is shown below in Figure 4.6-3.

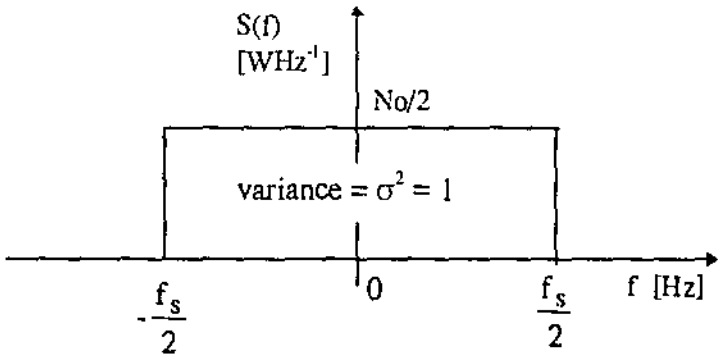

Figure 4.6-3: The power spectral density of bandlimited AWGN [7].

The Noise Regulator block is essentially an amplifier with an adjustable gain. In passing the AWGN noise through this block, the noise is multiplied by a factor of K. The effect of this is to increase the noise power by a factor of K^2. The power spectral density of the regulated AWGN then becomes that shown in Figure 4.6-4.

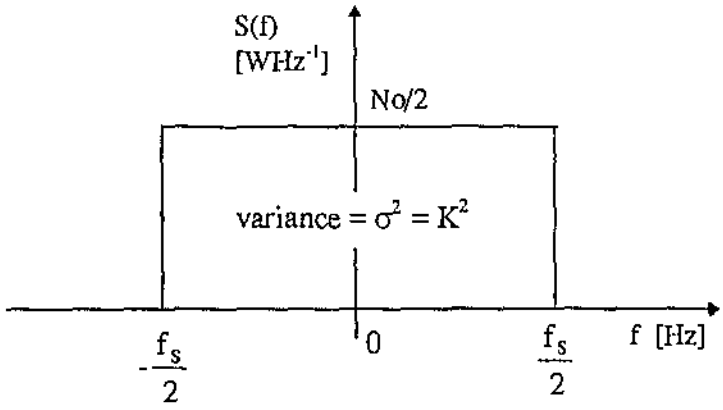

Figure 4.6-4: The power spectral density of the regulated bandlimited AWGN.

For simulation purposes, it is important to be able to control the intensity of the power spectral density. This is the most effective measure of noise intensity in the channel because it represents the normalisation of the noise power across the bandwidth of interest. The noise power spectral density No/2 can be expressed as:

$$\frac{No}{2} = \frac{\sigma^2}{f_s} = \frac{K^2}{f_s} \qquad [WHz^{-1}] \qquad (4.6\text{-}1)$$

Conversely, if a desired level of noise is required and is specified in terms of the power spectral density, the value of K for noise regulator is determined by:

$$K = \sqrt{\frac{No}{2} f_s} \qquad (4.6\text{-}2)$$

For the composite Rician channel of Simulator 3, the AWGN is regulated in this manner for the blocks "Random Number 3" and "Noise Regulator 3" which are illustrated in Figure 4.6-2. However, two random number generators are used to create the Rician channel. As specified in section (3.2), each one of these generators must generate independently distributed random variables (ζ and ξ) with identical probability density functions of zero mean and variance equal to the local mean reflected power q_s. This statement suggests that $\zeta\cos(2\pi f_c t)$ and $\xi\sin(2\pi f_c t)$ have a total combined power of q_s

and the random variables ζ and ξ have a power of q_s each. To do so, the gain of the amplifiers regulating the random variables ζ and ξ must be equal to $\sqrt{q_s}$. This is the value of the gain used in Regulators 1 and 2. Hence, the intensity of the multipath effect can be specified by q_s in the parameter file which is in turn enforced by Regulators 1 and 2. It is important to note that the random number generator operates at a fixed rate equivalent to the sampling rate of the simulator. Thus the rate of amplitude changes due to the multipath effect is very rapid. Consequently with this channel implementation, the multipath fading is fast. Also, because of the relatively large bandwidth of the random variables, the nature of the fading is frequency non-selective. To facilitate slow multipath fading in this implementation, SIMULINK's pre-built Bandlimited AWGN generator must be used. With this block, it is possible to reduce the rate of random number generation to obtain the desired rate of fading. Since this block is a recent addition to SIMULINK's library and was not available on the faster computers, slow multipath fading was not implemented.

The effect of the quadrature receiver is quite fundamental. It basically extracts the in-phase and quadrature components of the received signal, as was explained earlier by Figure 2.7-1. The filters used in the quadrature receiver cause delay and distortion to the in-phase and quadrature signal components. The delay is caused from the fact that these filters are digital FIR filters which are implemented with delay elements. This delay models the effect of a real filter. However, if these delays are not desired, the correlator can be specified to wait for a few sample intervals before commencing correlation. These delays are exact by nature of the FIR filter and SIMULINK, so it can be compensated effectively.

4.7 Analysis Tools

4.7.1 Debugging Tools

The debugging tools that were developed served well to clean up the bugs in the simulators during the implementation stage. After the simulator worked correctly, the

debugging tools helped to better understand the behaviour of CPM schemes being simulated.

Two of the most important debugging tools developed are called CHKTRAV( ) and COMPCOMP( ). These tools are MATLAB functions which can be executed from the MATLAB command line when the simulation is paused or completed. CHKTRAV( ) determines the behaviour of the transmitted signal. It determines the states traversed by the signal, the corresponding CCTAB position traversed, and the waveform lookup table version of the transmitted signal. To check that the correct states have been detected, the person debugging the program can view the contents of STATESTRAV and compare it with the equivalent form which was determined by the Viterbi detector called VSTATEPROGTAB. The CCTAB positions traversed is used to provide the CM that should have been obtained. By checking that the CMs are close to one when there is no noise and no interference, the correlator can be verified. Finally, the waveform lookup table version of the transmitted signal is used to ensure that the lookup table has been correctly generated.

COMPCOMP( ) plots the in-phase and quadrature components of the waveform lookup table against the received in-phase and quadrature components. A delay between the lookup table and received versions would most likely indicate that the filter delay has not been adequately compensated at the cross correlator module. In addition, a mismatch in the shape of the waveforms could be linked to a problem at the quadrature receiver.

4.7.2 Bandwidth Plotter

The bandwidth plotter is a MATLAB function which was developed to plot the spectral occupancy of a simulated CPM scheme. It is used in conjunction with Simulator 1 to conduct the spectral analysis of various CPM schemes.

Generally, the one sided bandwidth of the CPM signal falls in between the origin f = 0 and half the sampling frequency. In fact, the spectral occupancy is centred about the carrier frequency. This is illustrated ideally as

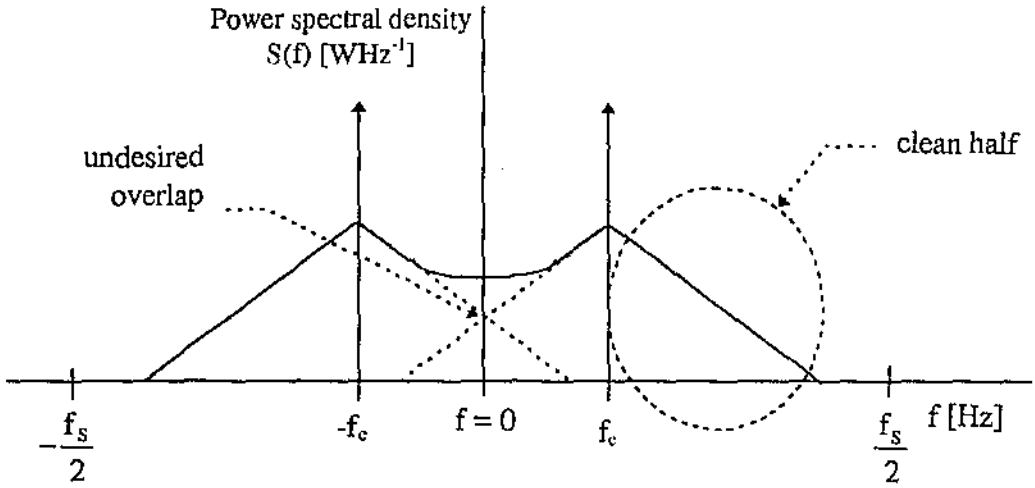

Figure 4.7.2: Power spectral density of a typical CPM signal showing a possible overlap of bandwidths which could result in incorrect measurements.

The power spectral density of the received bandpass signal can be plotted using MATLAB's PSD( ) which plots the power spectral density of the waveform for the positive frequencies. When plotting the power spectral density, it is important to check that there is no overlap between the mirrored bandwidths near the origin. However, when the carrier is chosen to be relatively close to the origin, this problem may not be avoided. Hence, to minimise the chance of incorrect measurements, the bandwidth plots are always conducted on the clean half of the signal bandwidth, that is $f_c \leq f \leq f_s / 2$.

The help facility in MATLAB provides the following assistance for PSD( ):

PSD Power Spectral Density estimate

Pxx = PSD(X,NFFT,Fs,WINDOW) estimates the Power Spectral Density of signal vector X using Welch's averaged periodogram method. X is divided into overlapping sections, each of which is detrended, then windowed by the WINDOW parameter, then zero-padded to length NFFT. The magnitude squared of the length NFFT DFTs of the sections are averaged to form Pxx. Pxx is length NFFT/2+1 for NFFT even, (NFFT+1)/2

for NFFT odd, or NFFT if the signal X is complex. If you specify a scalar for WINDOW, a Hanning window of that length is used. Fs is the sampling frequency which doesn't effect the spectrum estimate but is used for scaling of plots.

[Pxx,F] = PSD(X,NFFT,Fs,WINDOW,NOVERLAP) returns a vector of frequencies the same size as Pxx at which the PSD is estimated, and overlaps the sections of X by NOVERLAP samples.

The command used in the bandwidth plotter to perform obtain the power spectral density is

psdVECTOR = psd(VECTOR,1024,(1/ps),hanning(round(sizeVECTOR/5)),round((sizeVECTOR/5)/2))

where VECTOR stores the CPM signal, ps is the sampling interval, and sizeVECTOR stores the size of VECTOR. Thus, the power spectral density of the CPM signal is calculated using five overlapping hanning windows where the overlap is half the window size. psdVECTOR is also one thousand and twenty four samples long.

The desired "clean" region is then extracted from psdVECTOR and is then plotted in decibels. Also, the frequency on the x axis of the plot is normalised to the symbol rate. Samples of these plots are shown in section 5.

4.7.3 Monte Carlo Bit Error Simulation Regulator

This section describes the implementation of the Monte Carlo simulation regulator in Simulator 2 and 3. The Monte Carlo simulation technique is required to obtain a good estimate of the bit error rate performance of the CPM schemes for various intensities of noise and interference. The main feature of the Monte Carlo simulation regulator implemented in this project is that it is capable of automatically monitoring an entire simulation across the complete range of signal-to-noise ratios to be examined.

The term 'Monte Carlo' is merely a name for the implementation of a sequence of Bernoulli trials where the number of successes is counted and divided by the number of

trials. In the case of BER simulation, the Monte Carlo simulation regulator is counting errors rather than successes. Figure 4.7.3-1 shows the implementation of the Monte Carlo simulation regulator for the purpose of simulating a communication system. Shown as the "estimation procedure block", the simulation regulator is concerned only with the output of the source and the decision device's version of the original data. In particular, no assumptions about the input processes or the system is required. Since the source output is known, a comparison with the decoder's version will provide the error rate empirically [4].

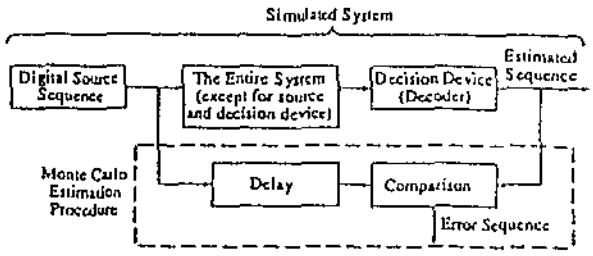

Figure 4.7.3-1: Schematic representation of implementation of Monte Carlo estimation procedure [4].

Since a delay between the input and output exist in a real system, it is important that the regulator synchronises the data sequences before comparison. Although the Monte Carlo simulation has been defined in essence, it is important to know the distribution of the decision variable. In other words, the accuracy of this technique needs to be known also. It is shown in [4] that as the number of trials $N \rightarrow \infty$, the estimator $\hat{e}$ tends to a normal distribution with mean e and variance e(1 - e)/N. Also, a confidence interval was derived which is plotted in Figure 4.7.3-2 showing the 90%, 95% and 99% confidence and the number of symbols required to achieve them.

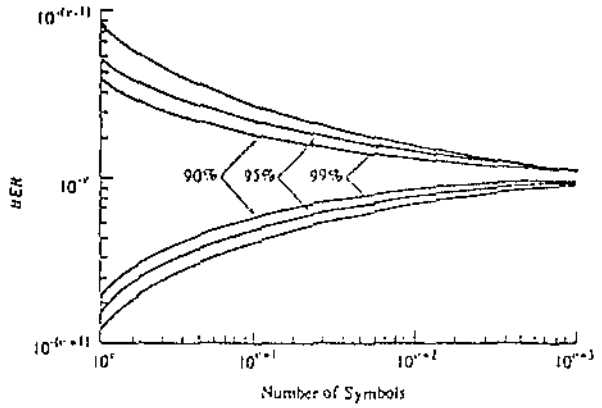

Figure 4.7.3-2: Confidence bands on BER when observed value is 10^{-v} for Monte Carlo technique based on the normal approximation [4].

The above Figure shows that for a particular bit error rate desired, the number of symbols taken to obtain the bit error rate will determine the estimator's accuracy. More importantly, it shows the number of symbols required to obtain 90%, 95% and 99% confidence in the estimator.

As a popular rule-of-thumb, N should be on the order of 10/e where e is the desired bit error. This coincides with a vertical slice at $N = 10^{v+1}$, which produces a 95% confidence interval [4].

The Monte Carlo simulation regulators developed in this project for Simulator 2 and 3 are based on an alternative perspective of the rule-of-thumb described in the last paragraph. The alternative perspective is that *ten errors need to be found to obtain a particular bit error rate with the confidence of 95%*. For a fixed power CPM signal, the regulator will initiate the simulation for a particular AWGN intensity. The simulator will then wait until 10 errors are detected before calculating the bit error rate for that signal-to-noise ratio. Once this is done, it will increase the noise intensity by a fixed increment to determine the bit error rate of the next signal-to-noise ratio. This process will continue until the lowest desired bit error is obtained or exceeded. When this happens, the regulator will trigger the stop simulation block to terminate the simulation. Each bit error rate obtained for a particular signal-to-noise ratio corresponds to a point on the bit error

performance plot. Hence, for a finer plot, it may be necessary to set the signal-to-noise ratio increments to a smaller value.

It is important to note that the bit error performance plots are plotted against *Eb/No per bit* where Eb is the energy of the CPM signal and No is twice the power spectral density of the AWGN. The energy of the CPM signal is given by $Eb = 0.5A^2T/n$ where A is the amplitude of the CPM signal, T is the symbol interval, and n is the number of bits per symbol. Since the power spectral density of the AWGN is a flat spectrum spanning the sampling bandwidth, the power spectral density of the additive noise No/2 is a better indicator of the noise intensity compared with the noise power K^2. This is because No defines the intensity of interference in the same spectral region as the bandwidth of the narrowband CPM signal.

For Simulator 3, every Monte Carlo simulation is performed for a fixed intensity of multipath scattering. Essentially, the bit error plot is a function of AWGN intensity. However, several of these plots are normally obtained for decreasing intensities of multipath scattering so that the effects of multipath scattering can be characterised. For each of these plots, the ratio Eb/Ss is clearly specified where Eb is the energy of the line-of-sight CPM signal while Ss is the power spectral density of the scattering component. Ss is a better indicator of the scattering intensity compared with q_s because it defines the intensity of interference in the same spectral region as the bandwidth of the narrowband, direct-line-of-sight CPM signal. This statement is true because the scattering component has a flat bandwidth spanning the sampling bandwidth.

The delay caused by the construction of the initial SWP in the Viterbi detector complicated the design of the regulator. This introduced some delay which the regulator has to accommodate. Also, when a new signal-to-noise ratio is introduced for another point on the bit error performance plot, the sliding window of the Viterbi detector had to be flushed. This meant waiting for approximately Nt symbol intervals while the new CM metrics were added into the sliding window.

The SIMULINK schematic for the Monte Carlo simulation regulator is illustrated in the following Figure. The signal line from the source is not required because the transmitted symbols are also stored as a global variable which can be accessed by the regulator block.

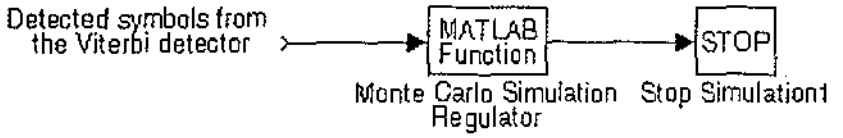

Figure 4.7-3: The SIMULINK schematic of the Monte Carlo simulation regulator and its connections.

4.8 Limitations

The major limitation of Simulators 2 and 3 is the processing speed. This is a problem when performing the Monte Carlo simulation for BER performance. For a reasonable CPM scheme, the BER performance plot to 10^{-3} can take a whole week of continuous processing on a workstation to generate. However, there is a possible solution to this problem. SIMULINK's C compiler is reputed to speed the performance of the simulator dramatically. Based on the performance figures provided by the manufacturer, it is very probable that the simulator performance can be increased substantially to make it useful for more demanding bit error performances.

The Eb/No ratio for the AWGN channel is measured before the quadrature receiver for the simulators implemented in this thesis in order to account for the influence of the quadrature receiver. It is important to consider the effects of the quadrature receiver since a bandpass model of the simulator was required to model the effects of multipath propagations according to the description given in [9] (refer to section 3.2). Although this approach offers a more realistic BER performance indication, it must be noted when comparing simulated BER performance results with literature results, that literature results often assume ideal filtering at the quadrature receiver. Thus, the simulated results

are likely to be slightly different to those commonly found in literature due to the non-ideal filtering at the quadrature receiver. However, this does not lessen the effectiveness of the simulator as a BER performance comparator for AWGN channels since the simulator is capable of generating a wide range of CPM schemes and so a relative comparison between the CPM schemes can be facilitated.

To support a direct performance comparison with literature results for AWGN channels, it is necessary to make certain changes to Simulator 2. Primarily, this involves converting the simulator into a baseband equivalent model. This can be easily done at the transmitter by not modulating the phase waveform with a carrier component and omitting the quadrature receiver [4]. Since the quadrature receiver is not required in the baseband equivalent model, the imperfections of the non-ideal filter can be avoided. However, it would not be suitable to use the baseband model for Simulator 3 because the multipath model described in [9] assumes bandpass transmission. Hence, it was decided that a bandpass model would be used in Simulator 2 so that it is exactly identical to Simulator 3, except without multipath fading. This way, the performance of Simulator 3 to varying intensities of AWGN without multipath fading can be first characterised using Simulator 2. The baseband equivalent model which will enable simulated results to be directly compared with most published results for AWGN channel is left as an avenue for future work.

Finally, the simulators implemented in this project is limited to fixed modulation index CPM schemes but it is possible to modify the simulators to accommodate multi-h CPM schemes without too much difficulty. However, it would be necessary to overcome the processing speed limitation first since multi-h schemes are more complicated to detect.

5 Simulation Trials

Preliminary results from the performed simulations are presented in this chapter. The results demonstrate the simulator's ability to assess the performance of CPM schemes in general. The simulations were performed under three performance categories:

1. Spectral occupancy
2. Bit error performance for the composite AWGN channel model
3. Bit error performance for the composite Rician channel model

The results from the performed simulations exhibit the characteristics of CPM schemes and coincided with the simulator's anticipated performance. The results also confirm that the simulator can be used to model the behaviour of CPM schemes in various channel environments.

Due to the time restriction on this project and the delayed arrival of the SIMULINK accelerator at the time this thesis was compiled, the BER performance plots were limited to a bit error rate of 10^{-3}. However, with the SIMULINK accelerator, the simulator can be easily left to process more demanding results within reasonable time.

5.1 Spectral Occupancy

Using Simulator 1, a modified simulator specifically designed to obtain the bandwidth performance of CPM signals, the following power spectral density plots were obtained for various CPM schemes. The exact procedure used to measure spectral occupancy is described in section 4.7.2. The power spectrum is plotted against the normalised frequency fT, where T is the inverse of the data symbol rate. In the following figures, 'M' denotes the modulation levels, 'h' the modulation index, 'L' the frequency pulse lengths and 'type' the pulse shaping type. The types of pulse shapes simulated include the unshaped rectangular (REC), Gaussian (G) and raised cosine (RC) pulses. Gaussian shaped pulses were generated according to the method outlined in Table 2.2-1 with the parameter $B_b = 2$.

By increasing the modulation levels of CPM schemes, the bandwidth of the signal is increased correspondingly. This relation is evident from Figure 5-1 for CPM schemes with h = 0.5, L = 3 and the rectangular pulse type. Clearly, larger modulation levels result in a wider bandwidth.

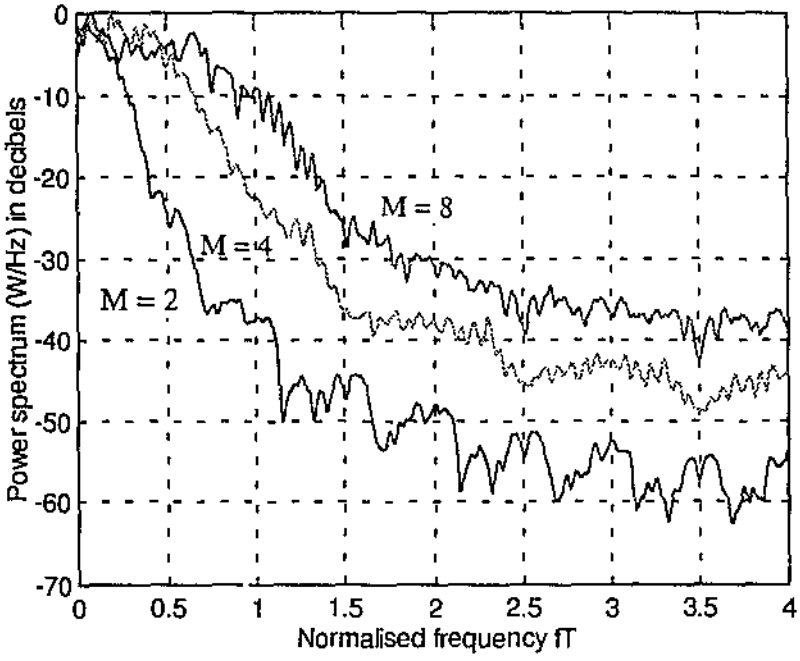

Figure 5-1: Power density spectrum of CPM schemes with different modulation levels where h = 0.5, L = 3, type = REC.

The effect of varying the modulation index of a CPM scheme where M = 4, L = 3 and the rectangular pulse type is employed is illustrated in Figure 5-2. It is evident that the modulation index h is directly proportional to the spectral occupancy of the signal.

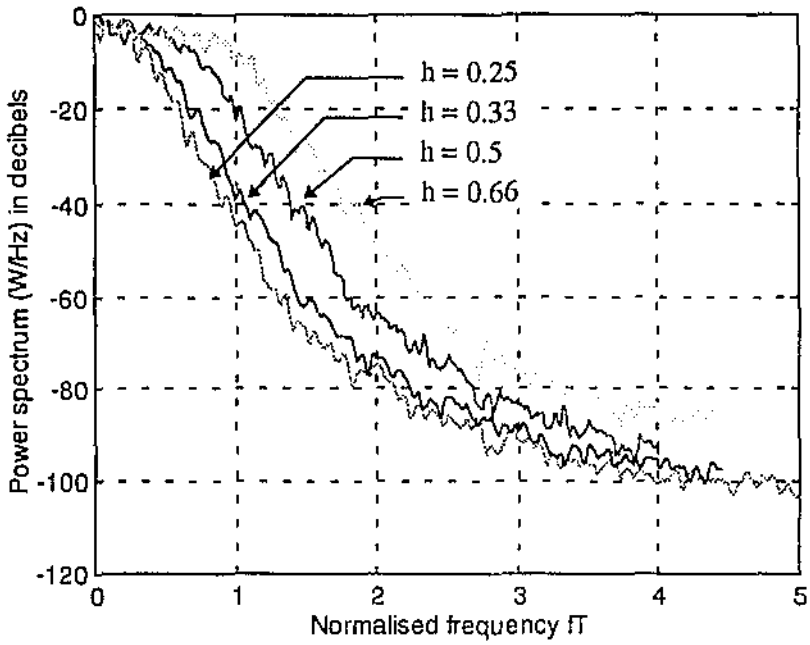

Figure 5-2: Power density spectrum of CPM schemes with different modulation indices where M = 4, L = 3, type = REC.

By increasing L, the pulse shape becomes smoother and the corresponding spectral occupancy of the signal is reduced. This is apparent from Figure 5-3 for the case where M = 2, h = 0.5, and the rectangular pulse type is employed.

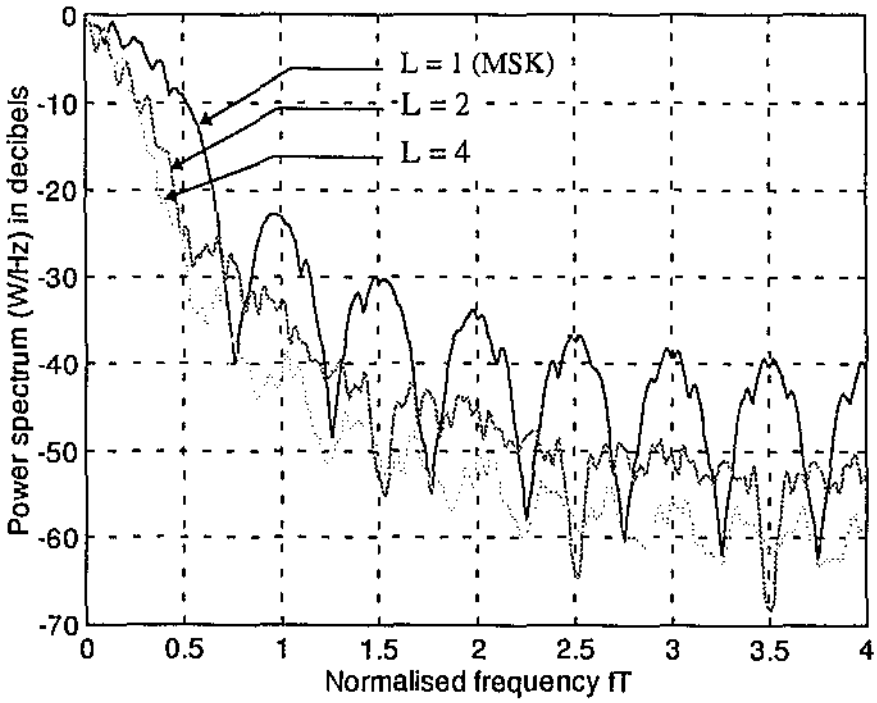

Figure 5-3: Power density spectrum of CPM schemes with different pulse lengths where M = 2, h = 0.5, type = REC.

The effect of using smooth pulses such as raised cosine and Gaussian shaping results in smaller bandwidth occupancy and hence greater bandwidth efficiency. This is observed in Figure 5-4 and Figure 5-5 where the overall spectral occupancy of the signal is reduced when pulse shaping is introduced. Compared with Figure 5-4, the larger L of Figure 5-5 results in a smoother pulse which subsequently reduces the corresponding spectral occupancy of the signal. Typically, longer pulse lengths are used together with pulse shaping to improve bandwidth utilisation. This is done because for an equivalent L, the unshaped rectangular pulse signal tends to have a smaller 20dB bandwidth when compared with the other two pulse shaped signals analysed. Thus, despite the reduced spectral occupancy of pulse shaped signals in general, larger values of L are used in conjunction with pulse shaping to provide a narrower bandwidth at frequencies closer to the carrier.

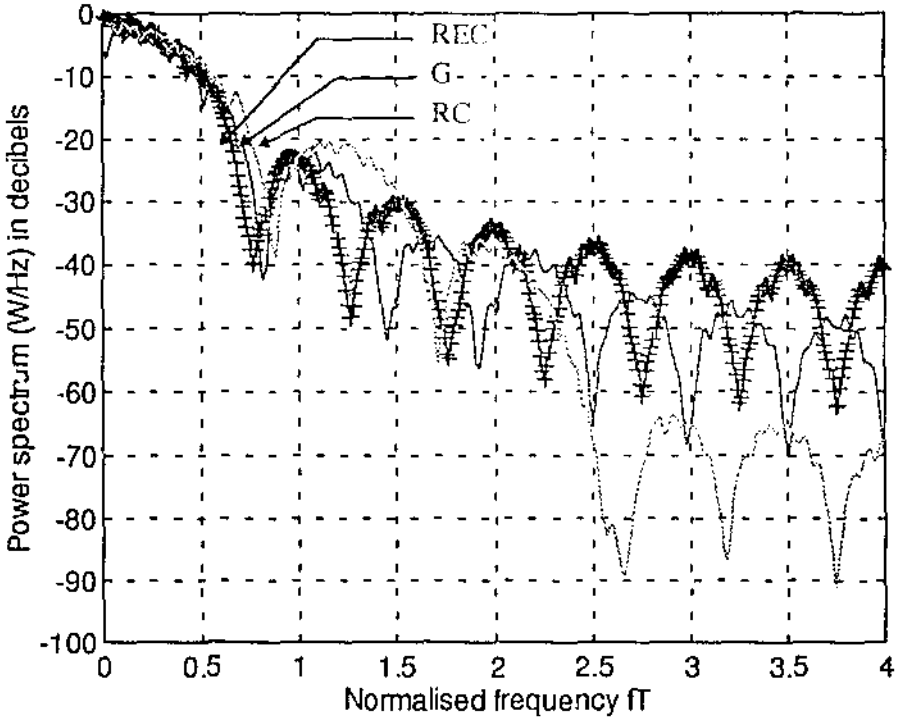

Figure 5-4: Power density spectrum of CPM schemes with different frequency pulse shapes where m = 2, L=1, h = 0.5.

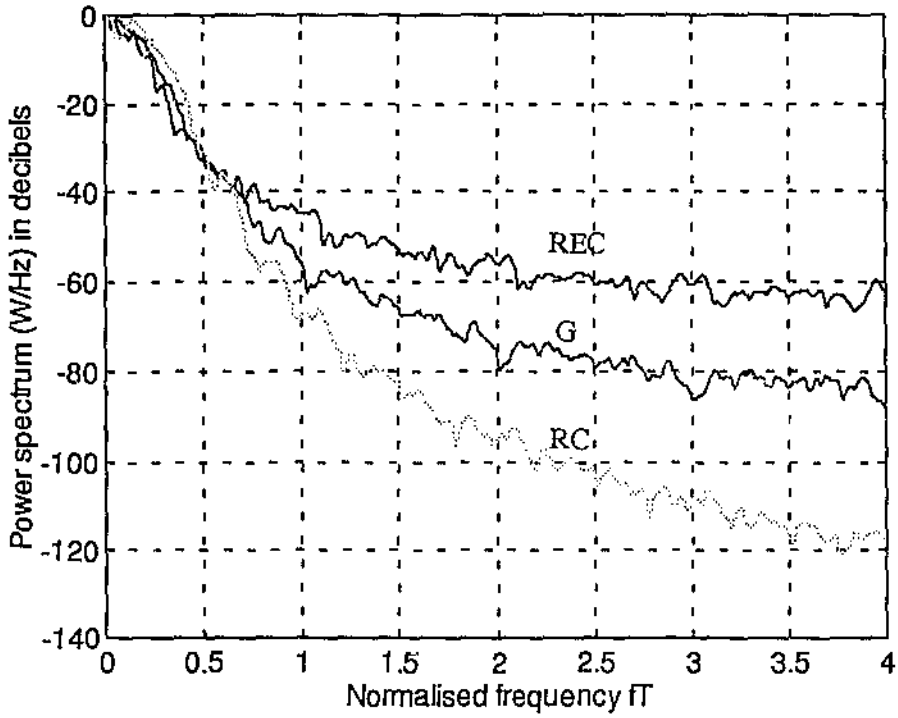

Figure 5-5: Power density spectrum of CPM schemes with different frequency pulse shapes where m = 2, L= 5, h = 0.5.

The spectral plots presented in this section for MSK match those in [1]. While the spectral performance for other modulation schemes are not available in the literatures reviewed, their comparative performance exhibit known CPM characteristics which were discussed in this chapter and confirmed in both [1] and [8].

5.2 Bit error performance for the composite AWGN channel model

Figure 5-6 shows the bit error (BER) performance of various CPM schemes to AWGN. The results were obtained using Simulator 2 where the composite AWGN channel is modelled. The probability of bit error was measured for varying strengths of Eb/No where Eb is the *energy per bit* of the information bearing signal and No/2 is the *power spectral density of the AWGN noise* added at the channel. The results were obtained in accordance to the Monte Carlo simulation method discussed earlier.

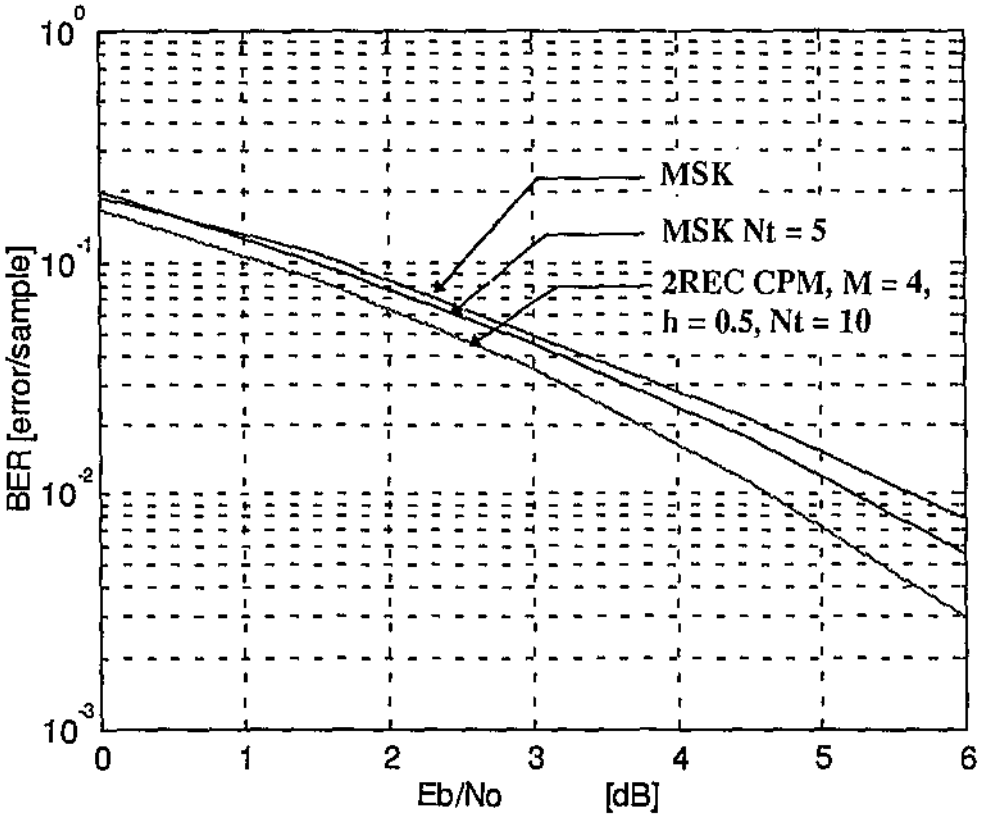

Figure 5-6: Comparison of the noise performance of different CPM schemes.

Increasing the length of the Viterbi detector improves the BER performance of CPM schemes. This characteristic is evident for the two MSK plots in Figure 5-6. Normally MSK has a Viterbi detector length of Nt = 2. Plotted on the same graph is the BER performance of MSK with a Viterbi detector length of Nt = 5. Clearly, the performance of the latter is slightly better. The third plot is the BER performance of 2REC quaternary CPM with h = 0.5 and Viterbi detector length Nt = 10. It is not surprising that the third

plot should have a significantly better BER performance since the minimum Euclidean distance of CPM schemes are increased by increasing either M, Nt, or L (section 2.8). The published BER performance of MSK [7] (refer to Figure 2.9-3) in an ideal channel with AWGN is marginally better than the simulated results by approximately 0.5dB at the horizontal axis. In comparing the simulated BER performance with published literature results, one must be aware of several important issues which are discussed in the following paragraphs.

For the more popular and less complicated CPM schemes, the BER performance to AWGN is usually derived theoretically and so does not consider the imperfections of filtering. Filtering is an essential part of quadrature reception and in the simulator that was used to obtain the results, the composite channel model included a non-ideal, fifth order, low-pass, digital filter which was used to isolate the bandpass components. The imperfect cut-off of a non-ideal, low pass filter is likely to introduce undesired frequency components just beyond the cut-off frequency and reduce the strength of desired signals just before the cut-off frequency. Also, in passing AWGN through a non-ideal filter, the output becomes 'coloured' [7]. In general, it is not easy to determine the exact distribution of a non-ideal filter output even if the distribution of the input signal is known [7].

The BER performance to AWGN of more complex CPM schemes are usually obtained through computer simulation and the simulation models used are typically baseband equivalent to avoid the complications of filtering at the quadrature receiver [4] (refer to section 4.8). Although this type of simulator is useful for comparing simulated performance with published literature results, it does not reflect the effects of quadrature reception which is an integral part of a real MLSE CPM system. The BER performance results obtained from the baseband equivalent simulator assumes that an ideal filter is used at the quadrature receiver. In the simulators implemented in this thesis, the BER results already account for the effects of quadrature reception since the Eb/No is added and measured just before the quadrature receiver. This is a more realistic approach.

Theoretical results for BER performance plots are obtained on the assumption that the probability of error is low [7]. This suggests that for lower Eb/No ratios, such as in the BER results obtained for this section, the analytical results are less accurate. In order to effectively compare simulated results with analytical results, the BER performance for larger Eb/No must be generated. This has not been feasible yet due to the processing speed limitation of the current simulator.

When sampling, the errors introduced from quantisation and digital signal processing are likely to affect the accuracy of the results. Usually, these errors are minimal. However, these effects should be carefully monitored to ensure reliable results.

The overall performance of a modulation scheme for a particular BER is determined by both its bandwidth utilisation and the required Eb/No. It is important to consider both metrics since a trade off exists between the two factors. To effectively compare the performance of a CPM scheme in an AWGN channel, its bandwidth utilisation and required Eb/No ratio for a particular BER rate must be plotted against the capacity boundary. The closer this point is to that boundary, the better its overall performance [7]. The procedure for this comparison was discussed earlier in section 2.9.

5.3 Bit error performance for the composite Rician channel model

Large multipath fading in a Rician channel degrades the quality of the received waveform so substantially that efforts to improve its quality by increasing the signal-to-noise ratio does little to enhance its BER performance.

Figure 5-7 illustrates the noise performance plots of a CPM scheme for varying intensities of multipath fading. Eb/Ss is the ratio of the *energy per bit of the direct component to the power spectral density of the scattered multipath components*. This ratio is inversely related to the intensity of multipath fading. The smaller this ratio, the larger the normalised power of the scattered multipath components when compared with the direct, line-of-sight component. Each line plot of Figure 5-7 was obtained while Eb/Ss was maintained constant. As in the case of AWGN channel, the probability of bit error was measured for varying strengths of Eb/No where Eb is the *energy per bit* of the information bearing signal and No/2 is the *power spectral density of the AWGN* added at the channel. The results were obtained in accordance to the Monte Carlo simulation method discussed in section 4.7.3.

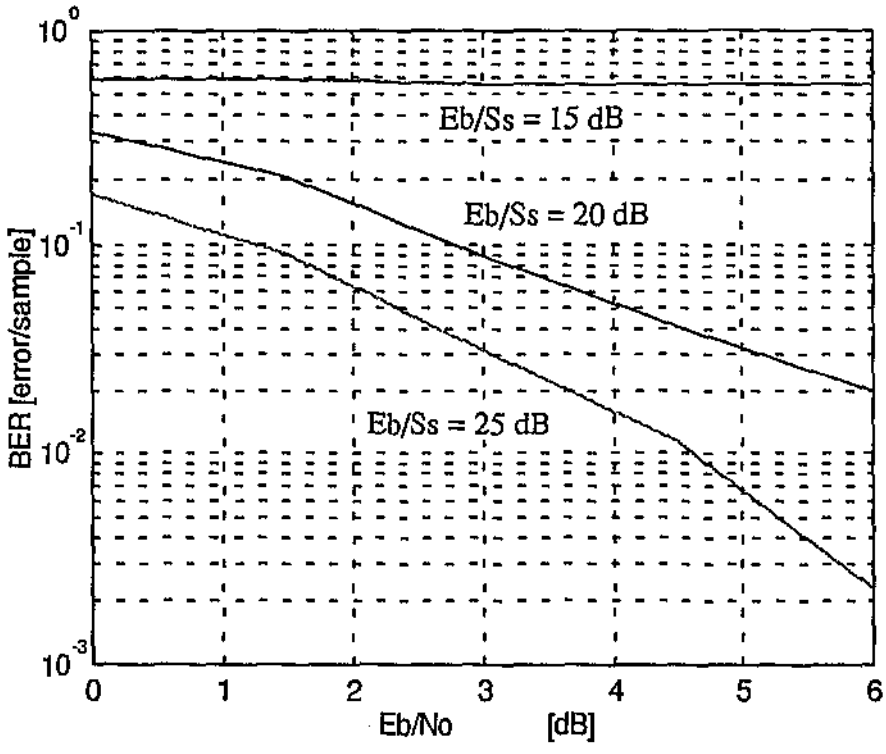

Figure 5-7: Noise performance of 2REC quaternary CPM, h = 0.5, Nt = 10 for fast-fading, frequency non-selective Rician channel.

Observing Figure 5-7, it is evident that large multipath fading has the effect of seriously dampening the BER performance. For the plot where Eb/Ss equals 15 dB, the multipath effect has degraded the received waveform so tremendously that increasing the Eb/No ratio had no effect in improving the BER performance of the CPM scheme. A significant improvement is made when the multipath interference is reduced by increasing the Eb/Ss ratio to 20 dB. With this, the BER performance becomes responsive to changes in Eb/No. As the multipath interference is reduced further, the BER performance becomes increasingly more responsive to increases in Eb/No.

In concluding this chapter, it is important to point out that some of the performance results from the conducted simulations may be available from published literature. However, the objective of conducting these simulations was not to regenerate already available information but to confirm the performance of a simulator which is capable of analysing a myriad of popular and unpopular CPM schemes.

From the preliminary trial of the simulator, from which the obtained results were presented in this chapter, it appears that the simulator is displaying known characteristics as discussed in [1] and [8]. It is difficult to directly compare the simulated results with those in the literature because the conditions upon which the literature results were obtained are slightly different. However, the presented results indicate that the simulator is performing correctly and is providing justifiable results which coincide with known characteristics.

6 Conclusions

The aim of the project reported in this thesis was to develop a flexible software simulator for analysing the performance of generic, non multi-h CPM schemes utilising Maximum Likelihood Sequence detection. The model should account for the effects of additive white Gaussian noise, reception filtering and multipath fading in an indoor wireless channel. This objective has been achieved, with performed simulations indicating that the simulator is performing correctly.

The most significant outcome of this project is that the performance of CPM schemes over an indoor microwave channel can now be characterised in order to determine the most suitable scheme for an indoor wireless communication system. A suitable CPM scheme should reinforce the anti-multipath properties of anti-multipath techniques such as CDMA and Frequency Hopping. This simulator has made this goal considerably closer by providing a test-bench from which a myriad of CPM schemes can be analysed in a realistic multipath environment.

The reason for building the simulator is that for most CPM schemes, their performances are not available from published literature. Moreover, for those schemes for which performances are available, the performances were usually not obtained for the environment of interest. Thus, it was essential to build a simulator which was capable of analysing CPM schemes in general for the specific environment of an indoor microwave communication system. As part of the literature survey conducted for this project, no published material employing such a comprehensive simulator was found. This simulator has met its specialised objective and it will provide data to supplement the gap in literature results.

As part of the implementation of this simulator in SIMULINK and MATLAB, a novel approach to integrating large and elaborate MATLAB functions with SIMULINK was devised. This method has effectively amalgamated the power of block diagram modelling in SIMULINK with the power of procedural language processing in MATLAB.

The opportunities opened by this project should be wisely exploited. Apart from providing the performance evaluation of CPM schemes over a wireless channel model, the simulator has other significant potentials. Developed in SIMULINK, the simulator may be used as a test bed from which an eventual hardware design can be evolved. Already, the simulator uses SIMULINK's pre-built hardware blocks together with the more complex program blocks. The gradual evolution of program blocks into hardware equivalent blocks in the design of an actual hardware implementation may be convenient for testing the newly designed hardware. Also, the modularity and realism of data flow modelling in SIMULINK allows additional modules to be effortlessly attached. The addition of an encoder and decoder will be useful for the study on the effects of encoding on CPM schemes over a real wireless channel. Further analysis into more complex multi-h CPM schemes can also be facilitated by making special adaptations to the existing simulator. Taken further, higher communication system levels can be constructed above the existing simulator to analyse the influence of schemes such as CDMA and Frequency Hopping on CPM over a real wireless channel. In short, the project has provided a stable platform from which further, more exciting work can be launched.

Just as the potential of CPM is limited by hardware complexity, the potential of this simulator is limited by the processing speed of SIMULINK. Currently, the simulator is too slow to process demanding BER performances. However, it is expected that the SIMULINK accelerator package will improve the performance significantly. A cross-compiler for SIMULINK is also available for cross-compilation to C for further processing speed improvement. Based on the specifications of these products, it is likely that the processing speed of the simulator can be improved substantially, making it possible to obtain more demanding BER performance evaluations.

CPM schemes are not generally popular in real world implementations because they are computationally intensive to detect. However, this will inevitably change when the ongoing trend of declining hardware prices will eventually make CPM systems more economically attainable. Another reason which would eventually increase the interest in more bandwidth-efficient modulation schemes such as CPM is that communication systems are increasingly more bandlimited due to the increasing demand for multimedia

and wireless communications. Not surprisingly, this simulator is reflecting all these trends by trying to find the most bandwidth efficient CPM scheme which is also robust enough to combat multipath effects in a wireless environment.

References

[1] Anderson, J. B., Aulin, T. & Sundberg, C., *Digital phase modulation*, Plenum Press: New York, 1986.

[2] Andrisano, O. & Ladisa, N., "On the spectral efficiency of CPM systems over real channel in the presence of adjacent channel and cochannel interference: a comparison between partial and full response systems", *IEEE Transactions on Vehicular Technology*, vol. 39, no. 2, pp 89-100, May 1990.

[3] Aulin, T., Rydbeck, N., & Sundberg, C. W., "Continuous phase modulation - Part II: Partial Response Signaling", *IEEE Transactions on Communications*, vol. COM-29, no. 3, pp 210-225, March 1981.

[4] Balaban, P., Jeruchim, M. C. & Shanmugan, K. S., *Simulation of communication systems*, Plenum Press: New York, 1992.

[5] Carlson, A. B., *Communication systems: An introduction to signals and noise in electrical communication*, Third Edition, McGraw-Hill Book Company: New York, 1986.

[6] Fonseka, J. P. & Mao, R., "Generalized nonlinear continuous phase modulation", *IEEE Transactions on Communications*, vol. 43, no. 5, May 1995.

[7] Haykin, S., *An introduction to analog and digital communications*, First Edition, John Wiley & Sons: New York, 1989.

[8] Proakis J. G., *Digital communications*, Third Edition, McGraw-Hill Book Company: New York, 1995.

[9] Linnartz, J. P., *Narrowband land-mobile radio networks*, Artech House Inc: Norwood, 1993.

[10] Peebles, P. Z., *Digital communications*, Prentice-Hall: New Jersey, 1987.

[11] Anonymous, *SIMULINK User's Guide*, December, The MathWorks Inc: Natick, 1993.

[12] Sundberg, C., "Continuous phase modulation", *IEEE Communications Magazine*, vol. 24, no. 4, pp 25-38, April 1986.

[13] Keng T. Tan, "COSSAP Simulation Model of DS-CDMA Indoor Microwave ATM LAN", *Honours Thesis Dissertation*, Department of Computer and Communication Engineering, Edith Cowan University, Western Australia: Australia, December 1995.

[14] Tsai, Y., Chang, J., "Using Frequency Hopping Spread Spectrum Technique to Combat Multipath Interference in a Multiaccessing Environment", *IEEE Transactions on Vehicular Technology*, vol. 43, no. 2, pp 211-222, May 1994.

[15] Schilling, T., *Principles of communication systems*, Prentice-Hall: New Jersey, 1989.

[16] Rappaport, T. S., Sandhu, S., "Radio-Wave Propagation for Emerging Wireless Personal-Communication Systems, *IEEE Antennas and Propagation Magazine*, vol. 36, no. 5, October 1994.

Appendix A: Simulator 1

bwplot.m

```
% FILE: bwplot.m
% ==============
% Plots the desired half bandwidth of a modulated signal, VECTOR.
% VECTOR should be either NOISELESSCPM or CPM when calling bwplot.m.
% To be executed after simbw.m simulation.

function bwplot(VECTOR)

% Input variables.
global ps rs fc;

% Transposing when required.
if size(VECTOR,2)<=2
 VECTOR=VECTOR';
 end;

psdVECTOR=psd(VECTOR);

% Complete P.S.D. of the modulated signal in dB.
TEMP1=(10*log(psdVECTOR/max(psdVECTOR)))/log(10);
TEMP1=TEMP1';

% Isolating the undesired section.
unwantedppt=fc/(1/(2*ps));

sizeTEMP1=size(TEMP1,2);

% The desired half of the bandwidth (right half b.w.).
HALFBW=TEMP1(1,round(unwantedppt*sizeTEMP1):sizeTEMP1);

sizeHALFBW=size(HALFBW,2);

T=[1:sizeHALFBW];
T=(T/sizeHALFBW)*(((1/(2*ps))-fc)/rs);

figure
plot(T,HALFBW)
```

inisim.m

```
% FILE: inisim.m
% ==============
% This is the parameter file for the initialisation
% of simbw.m prior to simulation.

% Note: SYMBOLS, PHASEWAVEFORM grow with duration of simulation.

% **** SIMULATOR SCOPE ****
% This simulator is capable of simulating CPM systems
% (constant amplitude, non multi-h).
% It can take MSK, CPFSK & CPM.
% h, the modulation index is such that: h>0;
% l, the length of phase pulse is such that: l>=1.

clear all;
close all;

% **** WORKSPACE VARIABLES ****
global AWGN CPM NOISELESSCPM;

% **** MODULATION INFORMATION ****
% Operational parameters
% ----------------------
% Symbol rate.
global rs;
% Sample period.
global ps;
% Length of pulse shape.
global l;
% Pulse shape type: 'LRC ', 'GMSK', 'LREC'.
global type;
% Signalling levels.
global m;
% Frequency deviation.
global df;
% Amplitude of CPM signal.
global a;
% Carrier frequency.
global wc fc;
% Noise power spectral density x 2.
global no;

rs=3;
ps=0.01;
l=3;
type='LRC ';
```

```
m=4;
df=1;
a=1;
fc=20;
no=0;

% Derived global parameters
% ------------------------
% FILES: phasepul.m possphas.m

% Modulation index.
global h;
% Symbol period.
global ts;
% Number of samples per symbol interval.
global sd;
% Symbol matrix.
global SYMMAT sizeSYMMAT;
% Phase pulse function.
global PHASEPUL;
global sizePHASEPUL sizePHASEPULLESS1;
% Set of possible phases.
global POSSPHAS;
global sizePOSSPHAS;
% The number of phase/symbol states.
global pml;
% Signal power.
global so;
% Noise regulator block's multiplier.
global mult;

h=(df/rs);
ts=1/rs;
sd=round(ts/ps);

SYMMAT=[];
n=0;
while (2*n+1)<=(m-1)
 SYMMAT=[-(2*n+1), SYMMAT, (2*n+1)];
 n=n+1;
 end;
sizeSYMMAT=size(SYMMAT,2);

phasepul;
sizePHASEPUL=size(PHASEPUL,2);
sizePHASEPULLESS1=sizePHASEPUL-1;
possphas;
sizePOSSPHAS=size(POSSPHAS,2);
```

```
pml=sizePOSSPHAS*(m^1);

% Signal power.
so=(0.5*a^2)/(log(m)/log(2));

% For noise regulator.
mult=sqrt(no);

% **** SYMBOL RECORDER ****
% BLOCK FILE: symbols.m

% Original data symbol sequence.
global SYMBOLS;
SYMBOLS=[];

% **** RANDOM SYMBOL GENERATOR ****
% BLOCK FILE: randsym.m

% Redeclaring all possible symbols.
global SYMMAT sizeSYMMAT;
% Current symbol, sample count per symbol interval.

global symbol randsymcnt;
% Number of samples per symbol interval.
global sd;

% Redefining SYMMAT.
% SYMMAT=[];
% n=0;
% while (2*n+1)<=(m-1)
%   SYMMAT=[-(2*n+1), SYMMAT, (2*n+1)];
%   n=n+1;
% end;
% Redefining number of samples per symbol interval.
% sd=round(ts/ps);

randsymcnt=1;

% **** PHASE PULSE GENERATOR ****
% BLOCK FILE: phasepg.m
% REDECLARED FILES: phasepul.m

% Redeclaring the phase pulse.
% global PHASEPUL sizePHASEPULLESS1;
% Redeclaring modulation index, number of samples per symbol interval.
% global h sd;

% SUM holds the added phase pulse sequence.
```

```
global SUM;
% Sample count per symbol, current position on the phase waveform.
global symbolcnt phasepgcnt;

% Redefining PHASEPUL.
% phasepul;
% sizePHASEPUL=size(PHASEPUL,2);
% sizePHASEPULLESS1=sizePHASEPUL-1;
% Redefining number of samples per symbol interval.
% sd=round(ts/ps);
% Redefining the modulation index.
% h=(df/rs);

SUM=[];
phasepgcnt=1;
symbolcnt=1;


% **** PHASEWAVEFORM ****
% BLOCK FILE: phasewf.m

% Transmitted phase.
global PHASEWAVEFORM;

% **** PHASE MODULATOR ****
% BLOCK FILE: None

% Redeclaring carrier frequency.
% global wc fc;
% Redeclaring sampling period.
% global ps;

% Carrier frequency in radians.
wc=2*pi*fc;
```

noisereg.m

```
% FILE: noisereg.m
% ================
% Regulates the intensity of noise into the AWGN channel.

function y=noisereg(x)
global mult;

y=mult*x;
```

phasepg.m

```
% FILE: phasepg.m
% ===============
% This function accepts the symbol sequence and outputs the
% superposition of the corresponding pulse shapes.

function y=phasepg(currentsym)

% Input variables.
% The phase pulse.
global PHASEPUL sizePHASEPULLESS1;
% Modulation index, number of samples per symbol interval.
global h sd phasepgcnt;

% Side-effect variables.
% Sample count per symbol, current position on the phase waveform.
global symbolcnt phasepgcnt;
% The sum of all phase pulses for the symbol sequence.
global SUM;

% Inserting the phase pulse into the SUM vector at every instance of a
% new symbol.
if symbolcnt==1
 % Increasing the size of SUM until there is enough room for the new phase pulse.
 while (phasepgcnt+sizePHASEPULLESS1)>size(SUM,2)
  SUM=[SUM,0];
  end;
 % Adding weighted pulses into SUM.
 for temp1=0:(sizePHASEPULLESS1)

SUM(phasepgcnt+temp1)=SUM(phasepgcnt+temp1)+2*pi*h*currentsym*PHASEPUL(
temp1+1);
  end;
 end;

y=SUM(phasepgcnt);

symbolcnt=symbolcnt+1;
if (symbolcnt>sd)
 symbolcnt=1;
 end;

phasepgcnt=phasepgcnt+1;
```

phasepul.m

```
% FILE: phasepul.m
% ================
% This function generates the pulse shaping function g(t)
% The types of pulse shapes that can be generated are
% LRC, GMSK, LREC.

% Undo this comment when not debugging.
function phasepul()

% For debugging purposes.
%l=3; % Memory.
%ps=0.01; % Sampling period.
%rs=3;
%type='LSRC'; % Type of modulation.

%ts=1/rs; % Symbol period.
%sd=round(ts/ps);

% Input variables.
% Undo this comment when not debugging.
global type l ts ps sd;

% Side-effect variables.
global PHASEPUL;

b=0.5; % The beta for LSRC type modulations.
bb=2; % The Bb for GMSK type modulations.
e=2.718281828;

if all(type=='LRC ')
  % T will always be l*sd long.
  TIME=0:ps:(l*sd-1)*ps;
  T=TIME;

  G=(1/(2*l*ts))*(1-cos((2*pi*T)/(l*ts)));

elseif all(type=='GMSK')
  % T will always be l*sd long.
  TIME=0:ps:(l*sd-1)*ps;
  TIME=TIME+0.00000001;
  EQNTIME=(TIME-(l*ts)/2)*(6/(l*ts))*(1/6);
  T=EQNTIME;

  t1=2*sqrt(2);
  t2=sqrt(2)*sqrt(log(2))/(2*pi*bb);
```

```
  Q1=(erfc((T-0.5/2)/t2))/t1;
  Q2=(erfc((T+0.5/2)/t2))/t1;

  G=(1/(2*0.5))*(Q1-Q2);

elseif all(type=='LREC')
  % T will always be l*sd long.
  T=0:ps:(l*sd-1)*ps;
  G=ones(1,l*sd)*1/(2*l*ts);
end;

if ~all(type=='LREC')
  % Zeroing G(1,1).
  G=G-G(1,1);
  end;

% Vertical scaling such that the height of cumsum(G) is 1/(2*ps).
cumsumG=cumsum(G);
G=G*(1/(2*ps))/cumsumG(1,l*sd);
cumsumG=cumsum(G);

% For debugging.
% close all;
% plot(G);
% figure
% plot(cumsumG);

PHASEPUL=G;
```

phasewf.m

```
% FILE: phasewf.m
% ==============
% This block stores the phase waveform into PHASEWAVEFORM.

function y=phasewf(x)

% Side-effect variable.
global PHASEWAVEFORM;

PHASEWAVEFORM=[PHASEWAVEFORM,x];
y=x;
```

possphas.m

```
% FILE: possphas.m
% ================
% This function generates all the possible phase states.

function possphas()

% For debugging purposes.
% df = 20; % The frequency distance between symbols.
% rs = 30; % The symbol rate.

% Input variables.
global df rs;

% Side-effect variable.
global POSSPHAS;

% Frequency deviation.
h=df/rs;

% h's greatest common denominator.
hgcd=gcd(df,rs);

% h=mm/p where mm and p are prime numbers.
mm=(df/hgcd);
p=(rs/hgcd);

% If mm is an even number.
if (rem(mm,2)==0)
 for count=0:(p-1)
  POSSPHAS(count+1)=count*pi*mm/p;
  end;
 end;

% If mm is an odd number.
if (rem(mm,2)==1)
 for count=0:(2*p-1)
  POSSPHAS(count+1)=count*pi*mm/p;
  end;
 end;
```

randsym.m

```
% FILE: randsym.m
% ==============
% This function generates the SYMMAT symbols randomly.

function y=randsym(x)

% Input variables.
% All possible symbols.
global SYMMAT sizeSYMMAT;

% Side-effect variables.
% Current symbol, sample count per symbol interval, number of samples per symbol
interval.
global symbol randsymcnt sd;

if (randsymcnt==1)
 symbol=SYMMAT(ceil(rand(1,1)*sizeSYMMAT));
 end;

y=symbol;
randsymcnt=randsymcnt+1;

% Changing the symbols.
if (randsymcnt>sd)
 randsymcnt=1;
 end;
```

simbw.m

```
% FILE: simbw.m
% ============
% SIMULINK file for Simulator 1: bandwidth measurements of CPM schemes.

function [ret,x0,str,ts,xts]=sim(t,x,u,flag);
%SIM is the M-file description of the SIMULINK system named SIM.
%       The block-diagram can be displayed by typing: SIM.
%
%       SYS=SIM(T,X,U,FLAG) returns depending on FLAG certain
%       system values given time point, T, current state vector, X,
%       and input vector, U.
%       FLAG is used to indicate the type of output to be returned in SYS.
%
%       Setting FLAG=1 causes SIM to return state derivatives, FLAG=2
%       discrete states, FLAG=3 system outputs and FLAG=4 next sample
```

```
%       time. For more information and other options see SFUNC.
%
%       Calling SIM with a FLAG of zero:
%       [SIZES]=SIM([],[],[],0), returns a vector, SIZES, which
%       contains the sizes of the state vector and other parameters.
%               SIZES(1) number of states
%               SIZES(2) number of discrete states
%               SIZES(3) number of outputs
%               SIZES(4) number of inputs
%               SIZES(5) number of roots (currently unsupported)
%               SIZES(6) direct feedthrough flag
%               SIZES(7) number of sample times
%
%       For the definition of other parameters in SIZES, see SFUNC.
%       See also, TRIM, LINMOD, LINSIM, EULER, RK23, RK45, ADAMS, GEAR.

% Note: This M-file is only used for saving graphical information;
%       after the model is loaded into memory an internal model
%       representation is used.

% the system will take on the name of this mfile:
sys = mfilename;
new_system(sys)
simver(1.3)
if (0 == (nargin + nargout))
   set_param(sys,'Location',[4,59,788,429])
   open_system(sys)
end;
set_param(sys,'algorithm',    'RK-45')
set_param(sys,'Start time',   '0.0')
set_param(sys,'Stop time',    '999999')
set_param(sys,'Min step size', 'ps')
set_param(sys,'Max step size', 'ps')
set_param(sys,'Relative error','1e-3')
set_param(sys,'Return vars',  '')


%   Subsystem 'Symbol Sequence'.

new_system([sys,'/','Symbol Sequence'])
set_param([sys,'/','Symbol Sequence'],'Location',[0,59,274,252])

add_block('built-in/Inport',[sys,'/','Symbol Sequence/x'])
set_param([sys,'/','Symbol Sequence/x'],...
                'position',[65,55,85,75])

add_block('built-in/S-Function',[sys,'/',['Symbol  Sequence/S-function',13,'M-file  which
plots',13,'lines',13,'']])
```

```
set_param([sys,'/',['Symbol Sequence/S-function',13,'M-file which plots',13,'lines',13,'']],...
                'function name','sfunyst',...
                'parameters','ax, color, npts, dt',...
                'position',[130,55,180,75])
add_line([sys,'/','Symbol Sequence'],[90,65;125,65])
set_param([sys,'/','Symbol Sequence'],...
                'Mask Display','plot(0,0,100,100,[83,76,63,52,42,38,28,16,11,84,11,11,11,90,90,11],[75,58,47,54,72,80,84,74,65,65,65,90,40,40,90,90])',...
                'Mask Type','Storage scope.')
set_param([sys,'/','Symbol Sequence'],...
                'Mask Dialogue','Storage scope using MATLAB graph window.\nEnter plotting ranges and line type.|Initial Time Range:|Initial y-min:|Initial y-max:|Storage pts.:|Line type (rgbw-.:xo):')
set_param([sys,'/','Symbol Sequence'],...
                'Mask Translate','npts = @4; color = @5; ax = [0, @1, @2, @3]; dt=-1;')
set_param([sys,'/','Symbol Sequence'],...
                'Mask Help','This block uses a MATLAB figure window to plot the input signal. The graph limits are automatically scaled to the min and max values of the signal stored in the scope''s signal buffer. Line type must be in quotes. See the M-file sfunyst.m.')
set_param([sys,'/','Symbol Sequence'],...
                'Mask Entries','5\/-10\/10\/200\/''y-/g--/c-./w:/m*/ro/b+''\/')

%    Finished composite block 'Symbol Sequence'.

set_param([sys,'/','Symbol Sequence'],...
                'position',[275,15,305,55])

add_block('built-in/MATLAB Fcn',[sys,'/','Phase Pulse Generator'])
set_param([sys,'/','Phase Pulse Generator'],...
                'MATLAB Fcn','phasepg',...
                'position',[295,133,355,167])

add_block('built-in/MATLAB Fcn',[sys,'/','SYMBOLS'])
set_param([sys,'/','SYMBOLS'],...
                'MATLAB Fcn','symbols',...
                'position',[260,78,320,112])

add_block('built-in/MATLAB Fcn',[sys,'/','Random Symbol Generator'])
set_param([sys,'/','Random Symbol Generator'],...
                'MATLAB Fcn','randsym',...
                'position',[130,128,190,162])

add_block('built-in/Step Fcn',[sys,'/','Step Initiator'])
set_param([sys,'/','Step Initiator'],...
```

```
                'Time','0',...
                'Before','1',...
                'position',[30,135,50,155])


%    Subsystem  ['Discrete-Time',13,'Integrator'].

new_system([sys,'/',['Discrete-Time',13,'Integrator']])
set_param([sys,'/',['Discrete-Time',13,'Integrator']],'Location',[392,641,765,859])

add_block('built-in/Inport',[sys,'/',['Discrete-Time',13,'Integrator/in_1']])
set_param([sys,'/',['Discrete-Time',13,'Integrator/in_1']],...
                'position',[25,70,45,90])

add_block('built-in/Sum',[sys,'/',['Discrete-Time',13,'Integrator/Sum']])
set_param([sys,'/',['Discrete-Time',13,'Integrator/Sum']],...
                'position',[150,63,180,127])

add_block('built-in/Unit Delay',[sys,'/',['Discrete-Time',13,'Integrator/Unit Delay']])
set_param([sys,'/',['Discrete-Time',13,'Integrator/Unit Delay']],...
                'Sample time','Ts',...
                'x0','X0',...
                'position',[210,85,260,105])

add_block('built-in/Gain',[sys,'/',['Discrete-Time',13,'Integrator/Gain']])
set_param([sys,'/',['Discrete-Time',13,'Integrator/Gain']],...
                'Gain','Ts',...
                'position',[80,61,110,99])

add_block('built-in/Outport',[sys,'/',['Discrete-Time',13,'Integrator/out_1']])
set_param([sys,'/',['Discrete-Time',13,'Integrator/out_1']],...
                'position',[320,85,340,105])
add_line([sys,'/',['Discrete-Time',13,'Integrator']],[115,80;145,80])
add_line([sys,'/',['Discrete-Time',13,'Integrator']],[50,80;75,80])
add_line([sys,'/',['Discrete-Time',13,'Integrator']],[185,95;205,95])
add_line([sys,'/',['Discrete-Time',13,'Integrator']],[265,95;315,95])
add_line([sys,'/',['Discrete-
Time',13,'Integrator']],[285,95;285,165;120,165;120,110;145,110])
set_param([sys,'/',['Discrete-Time',13,'Integrator']],...
                'Mask Display','dpoly(Ts,[1 -1],"z")',...
                'Mask Type','Discrete-Time Integrator',...
                'Mask Dialogue','Discrete-Time Integrator:|Initial Condition:|Sample
Time:')
set_param([sys,'/',['Discrete-Time',13,'Integrator']],...
                'Mask Translate','X0=@1;Ts=@2;',...
                'Mask Help','Implements a zeroth order discrete\nintegration using a gain,
a sum, and\na unit delay. Inputs may be scalar\nor vector.\n')
set_param([sys,'/',['Discrete-Time',13,'Integrator']],...
```

```
                'Mask Entries','0\/ps\/')


%    Finished composite block ['Discrete-Time',13,'Integrator'].

set_param([sys,'/',['Discrete-Time',13,'Integrator']],...
                'position',[375,129,415,171])

add_block('built-in/Sum',[sys,'/','Sum1'])
set_param([sys,'/','Sum1'],...
                'position',[625,155,645,175])

add_block('built-in/To Workspace',[sys,'/','To Workspace4'])
set_param([sys,'/','To Workspace4'],...
                'mat-name','AWGN',...
                'position',[625,22,675,38])

add_block('built-in/To Workspace',[sys,'/','To Workspace1'])
set_param([sys,'/','To Workspace1'],...
                'mat-name','CPM',...
                'position',[660,82,710,98])

add_block('built-in/MATLAB Fcn',[sys,'/','Noise Regulator'])
set_param([sys,'/','Noise Regulator'],...
                'MATLAB Fcn','noisereg',...
                'position',[530,53,590,87])

add_block('built-in/Gain',[sys,'/','Gain2'])
set_param([sys,'/','Gain2'],...
                'Gain','wc*ps',...
                'position',[250,203,280,247])

add_block('built-in/Sum',[sys,'/','Sum2'])
set_param([sys,'/','Sum2'],...
                'position',[340,220,360,240])

add_block('built-in/Unit Delay',[sys,'/','Unit Delay'])
set_param([sys,'/','Unit Delay'],...
                'Sample time','ps',...
                'position',[425,222,475,238])

add_block('built-in/Sum',[sys,'/','Sum3'])
set_param([sys,'/','Sum3'],...
                'position',[495,140,515,160])

add_block('built-in/MATLAB Fcn',[sys,'/','PHASEWAVEFORM'])
set_param([sys,'/','PHASEWAVEFORM'],...
                'MATLAB Fcn','phasewf',...
```

```
                'position',[440,88,500,122])

add_block('built-in/Gain',[sys,'/','Gain3'])
set_param([sys,'/','Gain3'],...
                'Gain','a',...
                'position',[580,168,610,212])

add_block('built-in/MATLAB Fcn',[sys,'/','cos'])
set_param([sys,'/','cos'],...
                'MATLAB Fcn','cos',...
                'position',[505,183,565,217])

add_block('built-in/White Noise',[sys,'/',['Random',13,'Number']])
set_param([sys,'/',['Random',13,'Number']],...
                'position',[460,12,505,48])

add_block('built-in/To Workspace',[sys,'/','To Workspace5'])
set_param([sys,'/','To Workspace5'],...
                'mat-name','NOISELESSCPM',...
                'buffer','2500',...
                'position',[590,283,680,297])


%    Subsystem ['Auto-Scale',13,'Graph'].

new_system([sys,'/',['Auto-Scale',13,'Graph']])
set_param([sys,'/',['Auto-Scale',13,'Graph']],'Location',[0,59,274,252])

add_block('built-in/Inport',[sys,'/',['Auto-Scale',13,'Graph/x']])
set_param([sys,'/',['Auto-Scale',13,'Graph/x']],...
                'position',[65,55,85,75])

add_block('built-in/S-Function',[sys,'/',['Auto-Scale',13,'Graph/S-function',13,'M-file
which plots',13,'lines',13,'']])
set_param([sys,'/',['Auto-Scale',13,'Graph/S-function',13,'M-file                which
plots',13,'lines',13,'']],...
                'function name','sfunyst',...
                'parameters','ax, color, npts, dt',...
                'position',[130,55,180,75])
add_line([sys,'/',['Auto-Scale',13,'Graph']],[90,65;125,65])
set_param([sys,'/',['Auto-Scale',13,'Graph']],...
                'Mask
Display','plot(0,0,100,100,[83,76,63,52,42,38,28,16,11,84,11,11,11,90,90,11],[75,58,47
,54,72,80,84,74,65,65,65,90,40,40,90,90])',...
                'Mask Type','Storage scope.')
set_param([sys,'/',['Auto-Scale',13,'Graph']],...
```

```
                'Mask Dialogue','Storage scope using MATLAB graph window.\nEnter
plotting ranges and line type.|Initial Time Range:|Initial y-min:|Initial y-max:|Storage
pts.:|Line type (rgbw-.:xo):')
set_param([sys,'/',['Auto-Scale',13,'Graph']],...
                'Mask Translate','npts = @4; color = @5; ax = [0, @1, @2, @3]; dt=-1;')
set_param([sys,'/',['Auto-Scale',13,'Graph']],...
                'Mask Help','This block uses a MATLAB figure window to plot the input
signal. The graph limits are automatically scaled to the min and max values of the signal
stored in the scope''s signal buffer. Line type must be in quotes. See the M-file
sfunyst.m.')
set_param([sys,'/',['Auto-Scale',13,'Graph']],...
                'Mask Entries','5\/-10\/10\/200\/''y-/g--/c-./w:/m*/ro/b+''\/')


%   Finished composite block ['Auto-Scale',13,'Graph'].

set_param([sys,'/',['Auto-Scale',13,'Graph']],...
                'position',[725,145,755,185])
add_line(sys,[360,150;370,150])
add_line(sys,[55,145;125,145])
add_line(sys,[195,145;210,145;210,35;270,35])
add_line(sys,[195,145;220,145;255,95])
add_line(sys,[325,95;335,95;335,120;275,120;275,150;290,150])
add_line(sys,[420,150;435,105])
add_line(sys,[650,165;660,165;660,140;625,140;625,90;655,90])
add_line(sys,[510,30;620,30])
add_line(sys,[510,30;515,30;525,70])
add_line(sys,[595,70;605,70;605,160;620,160])
add_line(sys,[285,225;335,225])
add_line(sys,[365,230;420,230])
add_line(sys,[480,230;495,230;490,255;320,255;320,235;335,235])
add_line(sys,[505,105;515,105;515,130;475,130;475,145;490,145])
add_line(sys,[480,230;490,155])
add_line(sys,[55,145;80,145;80,165;145,165;145,225;245,225])
add_line(sys,[520,150;530,150;530,165;485,165;485,200;500,200])
add_line(sys,[570,200;575,190])
add_line(sys,[615,190;620,170])
add_line(sys,[615,190;625,190;625,235;570,235;570,290;585,290])
add_line(sys,[650,165;720,165])

drawnow

% Return any arguments.
if (nargin | nargout)
        % Must use feval here to access system in memory
        if (nargin > 3)
                if (flag == 0)
                        eval(['[ret,x0,str,ts,xts]=',sys,'(t,x,u,flag);'])
```

```
            else
                eval(['ret =', sys,'(t,x,u,flag);'])
            end
        else
            [ret,x0,str,ts,xts] = feval(sys);
        end
else
        drawnow % Flash up the model and execute load callback
end
```

symbols.m

```
% FILE: symbols.m
% ==============
% This block stores the symbol stream into SYMBOLS.

function y=symbols(x)

% Side-effect variable.
global SYMBOLS;

SYMBOLS=[SYMBOLS,x];
y=x;
```

Appendix B: Simulators 2 and 3

compreg.m

```
% FILE: compreg.m
% ================
% Symbol comparator and Monte Carlo simulation regulator.

function y=compreg(x)

% **** SYMBOL COMPARATOR ****
% This block compares the transmitted symbols against the received symbols
% and records the number of errors.

% Comparison does not begin until (l+nt)sd+correldelay+1 samples have been
% received. Then, the comparison is with the (l-1)sd+1 sample transmitted.

% Input variables.
global tfirstsympos rfirstsympos diffofpos sd;
global SYMBOLS RSYMBOLS;

% Side-effect variables.
global errorcnt symcompcnt symcompsymcnt tenerrorcnt;

% **** SIMULATION REGULATOR ****
% This block is also capable of monitoring the simulation to obtain BER data.
% It will start the simulation with noise power equaled to the signal power.
% When 10 errors are obtained for the SNR, it will increment the SNR by dbinc
% until the best BER figures are obtained.
% Prior to changing the SNR, it will save the SNR and corresponding BER to a
% global variable and then flushing the Viterbi detector by waiting for nt+5
% symbol intervals.

% Input variables
global l a rs m Eb bestBER nt No bersim ps;

% Side-effect variables
global sgsymcnt tenerrorcnt SNRBER SNRperbit SNRperbitdB dbinc mult;

out=0;
symcompcnt=symcompcnt+1;
symcompsymcnt=symcompsymcnt+1;
if (symcompsymcnt>sd)
 symcompsymcnt=1;
 end;
```

```
if (symcompsymcnt==1) & (symcompcnt>rfirstsympos)
 sgsymcnt=sgsymcnt+1;

 % The start of every symbol greater than or equal to the starting instance.
 if RSYMBOLS(1,symcompcnt)~=SYMBOLS(1,diffofpos+symcompcnt)
  errorcnt=errorcnt+1;
  if (sgsymcnt>=1)
   tenerrorcnt=tenerrorcnt+1;
   end;
  end;

 if (sgsymcnt>=1)
  % The Viterbi decoder has been flushed.
  if (tenerrorcnt>=10) & (bersim==1)
   % The next ten errors have been counted.
   tenerrorcnt=0;
   sizeSNRBERpp=size(SNRBER,1);
   SNRBER(sizeSNRBERpp+1,1)=SNRperbitdB;
   BER=10/sgsymcnt;
   SNRBER(sizeSNRBERpp+1,2)=BER;
   SNRBER(sizeSNRBERpp+1,3)=sgsymcnt;
   SNRBER(sizeSNRBERpp+1,4)=mult;

   SNRperbitdB=SNRperbitdB+dbinc;
   SNRperbit=10^(SNRperbitdB/10);
   % Comment the following line for testing the flush.
   %mult=sqrt((a^2)/(4*rs*(log(m)/log(2))*SNRperbit));
   mult=sqrt(((a^2)*(1/ps))/(4*rs*(log(m)/log(2))*SNRperbit));

   % Flush out the Viterbi detector by ignoring nt+5 subsequent symbols.
   sgsymcnt=-(nt+l+5)+1;
   if (BER<=bestBER)
    % Stop the simulation if the BER reaches or is less than the best BER.
    out=1;
    end;
   end;
  end;
 end;

y=out;
```

correl.m

```
% FILE: correl.m
% ==============
% This function cross correlates the modulated signal with the expected
% waveforms derived from the phase functions of the ROM table.
% The input into this function are the inphase and quadrature components of
% the modulated signal.
% The output of this block is a timing signal used to synchronise the
% Viterbi detector. It is a "1" when the end of a symbol interval is reached.

function Y=correl(IQ)

% Input variables.
global COSWCT SINWCT COSWCTNPHASEFN CCTAB sd ps pml;

% Side-effect variables.
global correlcnt I Q;

correlcnt=correlcnt+1;

if correlcnt>sd
 correlcnt=1;
 end;

if (correlcnt>=1)
% At the initialisation of the simulation, correlcnt is given a negative
% value to delay the correlation of the received signal because of the delay
% in the digital filter.
 i=IQ(1,1);
 q=IQ(2,1);

% Store globally i and q.
I=[I,i];
Q=[Q,q];

 if (correlcnt==sd)
  % Calculating the unormalised cross correlation value.
  RT=(I.*COSWCT(1,:))-(Q.*SINWCT(1,:));
  TEMP1=corrcoef([RT',COSWCTNPHASEFN']);
  CC=TEMP1(2:(pml+1),1);

  CCTAB=[CCTAB,CC];
  I=[];
  Q=[];
  Y=[0.5,0.5];
 else
  Y=[0,0];
```

```
 end;
else
 Y=[0,0];
end;
```

inisimac.m

```
% FILE: inisimac.m
% ================
% The parameter file for Simulator 2: AWGN channel.
% To be executed before simulation.

% **** SIMULATOR SCOPE ****
% This simulator is capable of simulating CPM systems
% (constant amplitude, non multi-h).
% It can take MSK, CPFSK & CPM.
% h, the modulation index is such that: h>0;
% l, the length of phase pulse is such that: l>=1.
% nt, the length of the Viterbi processor is such that: nt>=1;

% Note: SYMBOLS, RSYMBOLS, VSTATEPROGTAB, CCTAB,
ACTUALPATHCCVAL
% PHASEWAVEFORM RINPHASE RQUAD SNRBER increases in size with
% duration of simulation.

clear all;
close all;

% **** WORKSPACE VARIABLES ****
global AWGN CPM RINPHASEUF RQUADUF NOISELESSCPM;

% **** MODULATION INFORMATION ****
% Operational parameters
% ----------------------
% Symbol rate.
global rs;
% Sample period.
global ps;
% Length of the phase pulse or the length of the frequency pulse function.
global l;
% Modulation type: 'LRC', 'GMSK', 'LREC'.
global type;
% Number of data symbols.
global m;
% Frequency deviation.
global df;
```

```
% Length of the Viterbi detector.
global nt;
% Amplitude of CPM signal.
global a;
% Carrier frequency.
global wc fc;

% Best BER desired.
global bestBER;
% Simulation decibel increment.
global dbinc;
% Do you want to run BER simulation? (Y->1, N->0)
% If bersim==0, no increments will be made to SNRperbit.
global bersim;
% Starting mult.
% startmult

rs=4;
ps=0.02;
l=1;
type='LREC';
m=2;
df=2;
% 2*l is a safe and recommended value for binary schemes.
nt=2;
a=10;
fc=10;

bestBER=0.001;
dbinc=3;
%bersim=0;
bersim=1;
startmult=3.5;

% Derived global parameters
% ------------------------
% FILES: phasepul.m possphas.m

% Modulation index.
global h;
% Symbol period.
global ts;
% Number of samples per symbol interval.
global sd;
% Symbol matrix.
global SYMMAT sizeSYMMAT;
% Phase pulse function.
global PHASEPUL;
```

```
global sizePHASEPUL sizePHASEPULLESS1;
% Set of possible phases.
global POSSPHAS;
global sizePOSSPHAS;
% The number of phase/symbol states.
global pml;

% Energy per bit.
global Eb;
% Noise power per hertz - Power spectral density.
global No;
% Signal to noise ratio per bit.
global SNRperbit;
% Signal to noise ratio per bit in dB.
global SNRperbitdB;
% Noise regulator block's multiplier.
global mult;

h=(df/rs);
ts=1/rs;
sd=round(ts/ps);

SYMMAT=[];
n=0;
while (2*n+1)<=(m-1)
 SYMMAT=[-(2*n+1), SYMMAT, (2*n+1)];
 n=n+1;
 end;
sizeSYMMAT=size(SYMMAT,2);

phasepul;
sizePHASEPUL=size(PHASEPUL,2);
sizePHASEPULLESS1=sizePHASEPUL-1;
possphas;
sizePOSSPHAS=size(POSSPHAS,2);
pml=sizePOSSPHAS*(m^l);

% Energy per bit.
Eb=(0.5*(a^2)*(1/rs))/(log(m)/log(2));

mult=startmult;

% Noise power per hertz - Power spectral density.
No=2*(mult^2);

if (No==0)
 SNRperbit=10^100;
 SNRperbitdB=1000;   % To prevent divide by zero error.
```

```
else
 SNRperbit=Eb/No;
 SNRperbitdB=10*(log(SNRperbit)/log(10));
end;

% **** SYMBOL RECORDER ****
% BLOCK FILE: symbols.m

% Original data symbol sequence.
global SYMBOLS;
SYMBOLS=[];

% **** RANDOM SYMBOL GENERATOR ****
% BLOCK FILE: randsym.m

% Redeclaring all possible symbols.
global SYMMAT sizeSYMMAT;
% Current symbol, sample count per symbol interval.

global symbol randsymcnt;
% Number of samples per symbol interval.
global sd;

% Redefining SYMMAT.
% SYMMAT=[];
% n=0;
% while (2*n+1)<=(m-1)
%   SYMMAT=[-(2*n+1), SYMMAT, (2*n+1)];
%   n=n+1;
% end;
% Redefining number of samples per symbol interval.
% sd=round(ts/ps);

randsymcnt=1;

% **** PHASE PULSE GENERATOR ****
% BLOCK FILE: phasepg.m
% REDECLARED FILES: phasepul.m

% Redeclaring the phase pulse.
% global PHASEPUL sizePHASEPULLESS1;
% Redeclaring modulation index, number of samples per symbol interval.
% global h sd;

% SUM holds the added phase pulse sequence.
global SUM;
% Sample count per symbol, current position on the phase waveform.
global symbolcnt phasepgcnt;
```

```
% Redefining PHASEPUL.
% phasepul;
% sizePHASEPUL=size(PHASEPUL,2);
% sizePHASEPULLESS1=sizePHASEPUL-1;
% Redefining number of samples per symbol interval.
% sd=round(ts/ps);
% Redefining the modulation index.
% h=(df/rs);

SUM=[];
phasepgcnt=1;
symbolcnt=1;


% **** PHASEWAVEFORM ****
% BLOCK FILE: phasewf.m

% Transmitted phase.
global PHASEWAVEFORM;

% **** PHASE MODULATOR ****
% BLOCK FILE: None

% Redeclaring carrier frequency.
% global wc fc;
% Redeclaring sampling period.
% global ps;

% Carrier frequency in radians.
wc=2*pi*fc;

% ***** QUADRATURE RECEIVER *****
% BLOCK FILES: None

% **** LOW PASS FILTER ****
% BLOCK FILES: None
% To filter out cos(2wct).
% This frequency is centred on 2*fc  (2*fc*2*ps = 4*fc*ps).
% The baseband component is inclusive up to ~2.5*rs  (2.5*ps*2*rs = 5*ps*rs).
% Filter cut-off is at 2.5*rs. Let fs/2=1 => fcutoff=2.5*rs*2*ps.

% Filter parameters.
global A B;

% Butterworth filter order.
filterorder=5;
% Changing the filter order requires changes to be made to the processing delay.
```

```
% Normally:
% [B,A]=butter(filterorder,2*ps*2*rs);
[B,A]=butter(filterorder,2.5*ps*2*rs);

% **** RINPHASE and RQUAD ****
% BLOCK FILES: rinphase.m, rquad.m
% Inphase component of the received signal.
global RINPHASE;
% Quadrature component of the received signal.
global RQUAD;

% **** CORRELATOR ****
% BLOCK FILE: correl.m

% Reclaring possible phases, number of samples in a symbol interval.
% global POSSPHAS sd;
% Redeclaring the number of possible state transitions, sampling period.
% global pml ps;
% Redeclaring phase pulse.
% global PHASEPUL;

% Cross correlation data table.
global CCTAB;
% State transition search table, state transition phase waveform table.
global SRCHTAB ROM;
% Sample count, in-phase component for the current interval, quadrature component.
global correlcnt I Q;
% Matrix of time increments.
global SYMTIMEMAT;
% Phase waveform of all state transitions (extracted from ROM).
global PHASEFNTAB;
% cos(wct), sin(wct), cos(wct + PHASEFNTAB).
global COSWCT SINWCT COSWCTNPHASEFN;

% Redefining POSSPHAS definition.
% possphas;
% Redefining number of samples per symbol interval.
% sd=round(ts/ps);
% Redefining the number of possible state transitions.
% pml=sizePOSSPHAS*(m^l);
% Redefining the phase function.
% phasepul;

srchtab;
romtab;

% Delay in the correlator processing to compensate for the delay caused
% by the digital filter.
```

```
% When delay is inserted, CCTAB have higher CC values.
correldelay=round(0.3*sd);
correlcnt=-correldelay+1;

PHASEFNTAB=ROM(:,2:(sd+1));

SYMTIME=[];
for count1=1:sd
 SYMTIME=[SYMTIME, count1*ps];
 end;

SYMTIMEMAT=[];
for count1=1:pml
 SYMTIMEMAT=[SYMTIMEMAT;SYMTIME];
 end;

COSWCT=cos(wc*SYMTIME);
SINWCT=sin(wc*SYMTIME);
COSWCTNPHASEFN=cos(wc*SYMTIMEMAT+PHASEFNTAB);

% **** VITERBI PROCESSOR ****
% BLOCK FILE:  viterbi.m.
% NON-BLOCK FILES: statetab.m, statetta.m.

% Redeclaring cross correlation data table.
% global CCTAB;
% Redeclaring: Shows all possible transitions and the corresponding symbols.
% global SRCHTAB;
% Redeclaring length of pulse, signalling levels, number of samples per symbol.
% global l m sd;
% Redeclaring length of the Viterbi detector.
% global nt;
% Redeclaring the transmitted symbol sequence record.
% global SYMBOLS;
% Redeclaring possible symbol matrix, possible phase matrix
% global SYMMAT POSSPHAS;

% List of all states.
global STATETAB;
global STATETABpp;
% List of all possible state transitions.
global STATETTAB;
% Surviving paths table.
global SPTAB;
% Surviving paths cross correlation table.
global SPCCTAB;
% Surviving paths symbol table.
global SPSYMTAB;
```

```
% Current symbol count, current symbol.
global vcnt vcurrentsymbol;
% Record of state transitions.
global VSTATEPROGTAB;
% Stores the first state when l=1 since there's no space in VSTATEPROG.
global VSTATEPROGTAB10;
% Stores the actual paths CC values.
global ACTUALPATHCCVAL;
% Lower bound for cummulative cross correlation value.
global mincumccval;

% For debugging purposes, FIRSTLSYMBOLS is global.
global FIRSTLSYMBOLS;

vcnt=0;
vcurrentsymbol=0;

% Redefining SRCHTAB.
% srchtab;
% Redefining POSSPHAS.
% possphas;
% Redefining SYMMAT.
% SYMTIME=[];
% for count1=1:sd
%   SYMTIME=[SYMTIME, count1*ps];
%   end;
% SYMTIMEMAT=[];
% for count1=1:pml
%   SYMTIMEMAT=[SYMTIMEMAT;SYMTIME];
%   end;

ACTUALPATHCCVAL=[];
statetab;
STATETABpp=size(STATETAB,1);
% STATETTAB is always sorted.
statetta;

mincumccval=-(nt*100);

% **** RECEIVED SYMBOL RECORDER ****
% FILE: rsymbols.m

% Received data symbol sequence.
global RSYMBOLS;

RSYMBOLS=[];

% **** SYMBOL COMPARATOR ****
```

```
% FILE: compreg.m

% Redeclaring.
% global l nt sd correldelay;
% Counts ten errors for simulation regulator block.
% global 10errorcnt;

global tfirstsympos rfirstsympos diffofpos;
global errorcnt symcompcnt symcompsymcnt;
global SYMBOLS RSYMBOLS;

% First symbol pos to be compared on SYMBOLS minus one.
% This symbol corresponds to the lth symbol transmitted.
tfirstsympos=(l-1)*sd;
% First symbol pos to be compared on RSYMBOLS minus one.
rfirstsympos=(l+nt)*sd+correldelay;

SYMBOLS=[];
RSYMBOLS=[];
errorcnt=0;
symcompcnt=0;
symcompsymcnt=0;
diffofpos=tfirstsympos-rfirstsympos;

% **** SIMULATION REGULATOR ****
% FILE: compreg.m

% Redeclaring.
% global so bestBER nt SNR dbinc no;
% Counts ten errors.
global tenerrorcnt;
% Records SNR against BER for the progress of the simulation.
global SNRBER;
% Symbol counter which is reset for each SNR.
global sgsymcnt;

tenerrorcnt=0;
SNRBER=[];
sgsymcnt=0;
```

inisimrc.m

```
% FILE: inisimrc.m
% ================
% The parameter file for Simulator 3: Rician & AWGN channel.
% To be executed before simulation.
```

```
% **** SIMULATOR SCOPE ****
% This simulator is capable of simulating CPM systems
% (constant amplitude, non multi-h).
% It can take MSK, CPFSK & CPM.
% h, the modulation index is such that: h>0;
% l, the length of phase pulse is such that: l>=1.
% nt, the length of the Viterbi processor is such that: nt>=1;

% Note: SYMBOLS, RSYMBOLS, VSTATEPROGTAB, CCTAB,
ACTUALPATHCCVAL
% PHASEWAVEFORM RINPHASE RQUAD SNRBER increases in size with
% duration of simulation.

clear all;
close all;

% Note: SYMBOLS, RSYMBOLS, VSTATEPROGTAB, CCTAB,
ACTUALPATHCCVAL
% PHASEWAVEFORM RINPHASE RQUAD SNRBER increases in size with
simulation.

% **** SIMULATOR SCOPE ****
% This simulator is capable of simulating CPM systems
% (constant amplitude, non multi-h).
% It can take MSK, CPFSK & CPM.
% h, the modulation index is such that: h>0;
% l, the length of phase pulse is such that: l>=1.
% nt, the length of the Viterbi processor is such that: nt>=1;

% **** WORKSPACE VARIABLES ****
global AWGN CPM RINPHASEUF RQUADUF NOISELESSCPM;

% **** MODULATION INFORMATION ****
% Operational parameters
% ----------------------
% Symbol rate.
global rs;
% Sample period.
global ps;
% Length of the phase pulse or the length of the frequency pulse function.
global l;
% Modulation type: 'LRC ', 'TFM ', 'LSRC', 'GMSK', 'LREC'.
global type;
% Number of data symbols.
global m;
% Frequency deviation.
global df;
```

```
% Length of the Viterbi detector.
global nt;
% Amplitude of CPM signal.
global a;
% Carrier frequency.
global wc fc;

% Best BER desired.
global bestBER;
% Simulation decibel increment.
global dbinc;
% Do you want to run BER simulation? (Y->1, N->0)
% If bersim==0, no increments will be made to SNRperbit.
global bersim;
% Starting mult.
% startmult
% Eb/Ss in dB - where Ss is the P.S.D x 2 of scattered components.
global SIRperbitdB;


rs=4;
ps=0.02;
l=2;
type='LREC';
m=4;
df=2;
% 2*l is a safe and recommended value for binary schemes.
nt=10;
a=1;
fc=10;

bestBER=0.001;
dbinc=3;
%bersim=0;
bersim=1;
startmult=1.25;
SIRperbitdB=15;


% Derived global parameters
% ------------------------
% FILES: phasepul.m possphas.m

% Modulation index.
global h;
% Symbol period.
global ts;
% Number of samples per symbol interval.
```

```
global sd;
% Symbol matrix.
global SYMMAT sizeSYMMAT;
% Phase pulse function.
global PHASEPUL;
global sizePHASEPUL sizePHASEPULLESS1;
% Set of possible phases.
global POSSPHAS;
global sizePOSSPHAS;
% The number of phase/symbol states.
global pml;

% Energy per bit.
global Eb;
% Noise power per hertz - Power spectral density.
global No;
% Signal to noise ratio per bit.
global SNRperbit;
% Signal to noise ratio per bit in dB.
global SNRperbitdB;
% Noise regulator block's multiplier.
global mult;
global multrc;

h=(df/rs);
ts=1/rs;
sd=round(ts/ps);

SYMMAT=[];
n=0;
while (2*n+1)<=(m-1)
 SYMMAT=[-(2*n+1), SYMMAT, (2*n+1)];
 n=n+1;
 end;
sizeSYMMAT=size(SYMMAT,2);

phasepul;
sizePHASEPUL=size(PHASEPUL,2);
sizePHASEPULLESS1=sizePHASEPUL-1;
possphas;
sizePOSSPHAS=size(POSSPHAS,2);
pml=sizePOSSPHAS*(m^l);

% Energy per bit.
Eb=(0.5*(a^2)*(1/rs))/(log(m)/log(2));

mult=startmult;
multrc=sqrt(((1/ps)*Eb)/(10^(SIRperbitdB/10)));
```

```
% Noise power per hertz - Power spectral density.
No=2*(mult^2)*ps;

if (No==0)
 SNRperbit=10^100;
 SNRperbitdB=1000;   % To prevent divide by zero error.
else
 SNRperbit=Eb/No;
 SNRperbitdB=10*(log(SNRperbit)/log(10));
end;

% **** SYMBOL RECORDER ****
% BLOCK FILE: symbols.m

% Original data symbol sequence.
global SYMBOLS;
SYMBOLS=[];

% **** RANDOM SYMBOL GENERATOR ****
% BLOCK FILE: randsym.m

% Redeclaring all possible symbols.
global SYMMAT sizeSYMMAT;
% Current symbol, sample count per symbol interval.

global symbol randsymcnt;
% Number of samples per symbol interval.
global sd;

% Redefining SYMMAT.
% SYMMAT=[];
% n=0;
% while (2*n+1)<=(m-1)
%   SYMMAT=[-(2*n+1), SYMMAT, (2*n+1)];
%   n=n+1;
% end;
% Redefining number of samples per symbol interval.
% sd=round(ts/ps);

randsymcnt=1;

% **** PHASE PULSE GENERATOR ****
% BLOCK FILE: phasepg.m
% REDECLARED FILES: phasepul.m

% Redeclaring the phase pulse.
% global PHASEPUL sizePHASEPULLESS1;
```

```
% Redeclaring modulation index, number of samples per symbol interval.
% global h sd;

% SUM holds the added phase pulse sequence.
global SUM;
% Sample count per symbol, current position on the phase waveform.
global symbolcnt phasepgcnt;

% Redefining PHASEPUL.
% phasepul;
% sizePHASEPUL=size(PHASEPUL,2);
% sizePHASEPULLESS1=sizePHASEPUL-1;
% Redefining number of samples per symbol interval.
% sd=round(ts/ps);
% Redefining the modulation index.
% h=(df/rs);

SUM=[];
phasepgcnt=1;
symbolcnt=1;


% **** PHASEWAVEFORM ****
% BLOCK FILE: phasewf.m

% Transmitted phase.
global PHASEWAVEFORM;

% **** PHASE MODULATOR ****
% BLOCK FILE: None

% Redeclaring carrier frequency.
% global wc fc;
% Redeclaring sampling period.
% global ps;

% Carrier frequency in radians.
wc=2*pi*fc;

% ***** QUADRATURE RECEIVER *****
% BLOCK FILES: None

% **** LOW PASS FILTER ****
% BLOCK FILES: None
% To filter out cos(2wct).
% This frequency is centred on 2*fc  (2*fc*2*ps = 4*fc*ps).
% The baseband component is inclusive up to ~2.5*rs  (2.5*ps*2*rs = 5*ps*rs).
% Filter cut-off is at 2.5*rs. Let fs/2=1 => fcutoff=2.5*rs*2*ps.
```

```
% Filter parameters.
global A B;

% Butterworth filter order.
filterorder=5;
% Changing the filter order requires changes to be made to the processing delay.
% Normally:
% [B,A]=butter(filterorder,2*ps*2*rs);
[B,A]=butter(filterorder,2.5*ps*2*rs);

% **** RINPHASE and RQUAD ****
% BLOCK FILES: rinphase.m, rquad.m
% Inphase component of the received signal.
global RINPHASE;
% Quadrature component of the received signal.
global RQUAD;

% **** CORRELATOR ****
% BLOCK FILE: correl.m

% Reclaring possible phases, number of samples in a symbol interval.
% global POSSPHAS sd;
% Redeclaring the number of possible state transitions, sampling period.
% global pml ps;
% Redeclaring phase pulse.
% global PHASEPUL;

% Cross correlation data table.
global CCTAB;
% State transition search table, state transition phase waveform table.
global SRCHTAB ROM;
% Sample count, in-phase component for the current interval, quadrature component.
global correlcnt I Q;
% Matrix of time increments.
global SYMTIMEMAT;
% Phase waveform of all state transitions (extracted from ROM).
global PHASEFNTAB;
% cos(wct), sin(wct), cos(wct + PHASEFNTAB).
global COSWCT SINWCT COSWCTNPHASEFN;

% Redefining POSSPHAS definition.
% possphas;
% Redefining number of samples per symbol interval.
% sd=round(ts/ps);
% Redefining the number of possible state transitions.
% pml=sizePOSSPHAS*(m^l);
% Redefining the phase function.
```

```
% phasepul;

srchtab;
romtab;

% Delay in the correlator processing to compensate for the delay caused
% by the digital filter.
% When delay is inserted, CCTAB have higher CC values.
correldelay=round(0.3*sd);
correlcnt=-correldelay+1;

PHASEFNTAB=ROM(:,2:(sd+1));

SYMTIME=[];
for count1=1:sd
 SYMTIME=[SYMTIME, count1*ps];
 end;

SYMTIMEMAT=[];
for count1=1:pml
 SYMTIMEMAT=[SYMTIMEMAT;SYMTIME];
 end;

COSWCT=cos(wc*SYMTIME);
SINWCT=sin(wc*SYMTIME);
COSWCTNPHASEFN=cos(wc*SYMTIMEMAT+PHASEFNTAB);

% **** VITERBI PROCESSOR ****
% BLOCK FILE:  viterbi.m.
% NON-BLOCK FILES: statetab.m, statetta.m.

% Redeclaring cross correlation data table.
% global CCTAB;
% Redeclaring: Shows all possible transitions and the corresponding symbols.
% global SRCHTAB;
% Redeclaring length of pulse, signalling levels, number of samples per symbol.
% global l m sd;
% Redeclaring length of the Viterbi detector.
% global nt;
% Redeclaring the transmitted symbol sequence record.
% global SYMBOLS;
% Redeclaring possible symbol matrix, possible phase matrix
% global SYMMAT POSSPHAS;

% List of all states.
global STATETAB;
global STATETABpp;
% List of all possible state transitions.
```

```
global STATETTAB;
% Surviving paths table.
global SPTAB;
% Surviving paths cross correlation table.
global SPCCTAB;
% Surviving paths symbol table.
global SPSYMTAB;
% Current symbol count, current symbol.
global vcnt vcurrentsymbol;
% Record of state transitions.
global VSTATEPROGTAB;
% Stores the first state when l=1 since there's no space in VSTATEPROG.
global VSTATEPROGTAB10;
% Stores the actual paths CC values.
global ACTUALPATHCCVAL;
% Lower bound for cummulative cross correlation value.
global mincumccval;

% For debugging purposes, FIRSTLSYMBOLS is global.
global FIRSTLSYMBOLS;

vcnt=0;
vcurrentsymbol=0;

% Redefining SRCHTAB.
% srchtab;
% Redefining POSSPHAS.
% possphas;
% Redefining SYMMAT.
% SYMTIME=[];
% for count1=1:sd
%   SYMTIME=[SYMTIME, count1*ps];
%   end;
% SYMTIMEMAT=[];
% for count1=1:pml
%   SYMTIMEMAT=[SYMTIMEMAT;SYMTIME];
%   end;

ACTUALPATHCCVAL=[];
statetab;
STATETABpp=size(STATETAB,1);
% STATETTAB is always sorted.
statetta;

mincumccval=-(nt*100);

% **** RECEIVED SYMBOL RECORDER ****
% FILE: rsymbols.m
```

```
% Received data symbol sequence.
global RSYMBOLS;

RSYMBOLS=[];

% **** SYMBOL COMPARATOR ****
% FILE: compreg.m

% Redeclaring.
% global l nt sd correldelay;
% Counts ten errors for simulation regulator block.
% global 10errorcnt;

global tfirstsympos rfirstsympos diffofpos;
global errorcnt symcompcnt symcompsymcnt;
global SYMBOLS RSYMBOLS;

% First symbol pos to be compared on SYMBOLS minus one.
% This symbol corresponds to the lth symbol transmitted.
tfirstsympos=(l-1)*sd;
% First symbol pos to be compared on RSYMBOLS minus one.
rfirstsympos=(l+nt)*sd+correldelay;

SYMBOLS=[];
RSYMBOLS=[];
errorcnt=0;
symcompcnt=0;
symcompsymcnt=0;
diffofpos=tfirstsympos-rfirstsympos;

% **** SIMULATION REGULATOR ****
% FILE: compreg.m

% Redeclaring.
% global so bestBER nt SNR dbinc no;
% Counts ten errors.
global tenerrorcnt;
% Records SNR against BER for the progress of the simulation.
global SNRBER;
% Symbol counter which is reset for each SNR.
global sgsymcnt;

tenerrorcnt=0;
SNRBER=[];
sgsymcnt=0;
```

noisereg.m

```
% FILE: noisereg.m
% ================
% Regulates the intensity of noise into the AWGN channel.

function y=noisereg(x)
global mult;

y=mult*x;
```

phasepg.m

```
% FILE: phasepg.m
% ================
% This function accepts the symbol sequence and outputs the
% superposition of the corresponding pulse shapes.

function y=phasepg(currentsym)

% Input variables.
% The phase pulse.
global PHASEPUL sizePHASEPULLESS1;
% Modulation index, number of samples per symbol interval.
global h sd phasepgcnt;

% Side-effect variables.
% Sample count per symbol, current position on the phase waveform.
global symbolcnt phasepgcnt;
% The sum of all phase pulses for the symbol sequence.
global SUM;

% Inserting the phase pulse into the SUM vector at every instance of a
% new symbol.
if symbolcnt==1
 % Increasing the size of SUM until there is enough room for the new phase pulse.
 while (phasepgcnt+sizePHASEPULLESS1)>size(SUM,2)
  SUM=[SUM,0];
  end;
 % Adding weighted pulses into SUM.
 for temp1=0:(sizePHASEPULLESS1)

SUM(phasepgcnt+temp1)=SUM(phasepgcnt+temp1)+2*pi*h*currentsym*PHASEPUL(
temp1+1);
  end;
 end;
```

```
y=SUM(phasepgcnt);

symbolcnt=symbolcnt+1;
if (symbolcnt>sd)
 symbolcnt=1;
 end;

phasepgcnt=phasepgcnt+1;
```

phasepul.m

```
% FILE: phasepul.m
% ================
% This function generates the pulse shaping function g(t)
% The types of pulse shapes that can be generated are
% LRC, GMSK, LREC.

% Undo this comment when not debugging.
function phasepul()

% For debugging purposes.
%l=4; % Memory.
%ps=0.02; % Sampling period.
%rs=4;
%type='GMSK'; % Type of modulation.

%ts=1/rs; % Symbol period.
%sd=round(ts/ps);

% Input variables.
% Undo this comment when not debugging.
global type l ts ps sd;

% Side-effect variables.
global PHASEPUL;

b=0.5; % The beta for LSRC type modulations.
bb=2; % The Bb for GMSK type modulations.
e=2.718281828;

if all(type=='LRC ')
 % T will always be l*sd long.
 TIME=0:ps:(l*sd-1)*ps;
 T=TIME*(0.8/1);
```

```
 G=(1/(2*l*ts))*(1-cos((2*pi*T)/(l*ts)));

elseif all(type=='GMSK')
 % T will always be l*sd long.
 TIME=0:ps:(l*sd-1)*ps;
 TIME=TIME+0.00000001;
 EQNTIME=(TIME-(l*ts)/2)*(6/(l*ts))*(0.5/6);
 T=EQNTIME;

 t1=2*sqrt(2);
 t2=sqrt(2)*sqrt(log(2))/(2*pi*bb);

 Q1=(erfc((T-0.5/2)/t2))/t1;
 Q2=(erfc((T+0.5/2)/t2))/t1;

 G=(1/(2*0.5))*(Q1-Q2);

elseif all(type=='LREC')
 % T will always be l*sd long.
 T=0:ps:(l*sd-1)*ps;
 G=ones(1,l*sd)*1/(2*l*ts);
end;

if ~all(type=='LREC')
 % Zeroing G(1,1).
 G=G-G(1,1);
 end;

% Vertical scaling such that the height of cumsum(G) is 1/(2*ps).
cumsumG=cumsum(G);
G=G*(1/(2*ps))/cumsumG(1,l*sd);
cumsumG=cumsum(G);

% For debugging.
% close all;
% plot(G);
% figure
% plot(cumsumG);

PHASEPUL=G;

%figure
%plot(PHASEPUL)
%figure
%plot(cumsum(PHASEPUL))
```

phasewf.m

```
% FILE: phasewf.m
% ==============
% This block stores the phase waveform into PHASEWAVEFORM.

function y=phasewf(x)

% Side-effect variable.
global PHASEWAVEFORM;

PHASEWAVEFORM=[PHASEWAVEFORM,x];
y=x;
```

possphas.m

```
% FILE: possphas.m
% ================
% This function generates all the possible phase states.

function possphas()

% For debugging purposes.
% df = 20; % The frequency distance between symbols.
% rs = 30; % The symbol rate.

% Input variables.
global df rs;

% Side-effect variable.
global POSSPHAS;

% Frequency deviation.
h=df/rs;

% h's greatest common denominator.
hgcd=gcd(df,rs);

% h=mm/p where mm and p are prime numbers.
mm=(df/hgcd);
p=(rs/hgcd);

% If mm is an even number.
if (rem(mm,2)==0)
  for count=0:(p-1)
    POSSPHAS(count+1)=count*pi*mm/p;
```

```
  end;
 end;

% If mm is an odd number.
if (rem(mm,2)==1)
 for count=0:(2*p-1)
  POSSPHAS(count+1)=count*pi*mm/p;
  end;
 end;


POSSPHAS=sort(rem(POSSPHAS,2*pi));
```

randsym.m

```
% FILE: randsym.m
% ===============
% This function generates the SYMMAT symbols randomly.

function y=randsym(x)

% Input variables.
% All possible symbols.
global SYMMAT sizeSYMMAT;

% Side-effect variables.
% Current symbol, sample count per symbol interval, number of samples per symbol
interval.
global symbol randsymcnt sd;

if (randsymcnt==1)
 symbol=SYMMAT(ceil(rand(1,1)*sizeSYMMAT));
 end;

y=symbol;
randsymcnt=randsymcnt+1;

% Changing the symbols.
if (randsymcnt>sd)
 randsymcnt=1;
 end;
```

rcreg.m

```
% FILE: rcreg.m
% ============
% This function regulates the intensity of multipath fading
% into the channel.

function y=rcreg(x)

global multrc;

y=multrc*x;
```

rinphase.m

```
% FILE: rinphase.m
% ==============
% This block stores the received inphase signal into RINPHASE.

function y=rinphase(x)

% Side-effect variable.
global RINPHASE;

RINPHASE=[RINPHASE,x];
y=x;
```

romtab.m

```
% FILE: romtab.m
% ============
% This function generates the phase function ROM for CPM modulations.
% The ROMs are divided into SRCHTAB and ROM which are both global vars.
% SRCHTAB is the search table to determine the state of the modulation.
% From SRCHTAB we determine the position of the phase function in the
% ROM table. SRCHTAB is generated by srchtab.m and ROM by romtab.m.

function romtab()

% ROM table holds the row numbers in the 1st column.

% For debugging purposes:
% ***** Function parameters. ***************
```

```
% type='TFM ';
% df=1;  % Frequency deviation.
% rs=2;  % Symbol rate.
% m=4;   % The number of symbols. Usually 2^n.
% ps=0.01;
% l=3;   % Length of phase function expressed as the number of symbol periods.
% ***************************************

% Input variables.
global type df rs m ps l;
global POSSPHAS SRCHTAB PHASEPUL;

% Side-effect variables.
global ROM;

h=(df/rs);
ts=(1/rs);
sd=round(ts/ps);

[sizeSRCHTABr, sizeSRCHTABc]=size(SRCHTAB);

% Integral of the frequency pulse function.
INTG=cumsum(PHASEPUL)*ps;

% Forming the ROM table.
for row=1:sizeSRCHTABr
 theta=SRCHTAB(row, 1);
 shift=0;
 SUMWINDOW=[];
 CURRENTWINDOW=[];
 SUMWINDOW=[zeros(1,l*sd)];
 CURRENTWINDOW=[zeros(1,l*sd)];
 for col=2:(sizeSRCHTABc-1)
  CURRENTWINDOW=[];
  CURRENTWINDOW=[zeros(1,l*sd)];

  % The following count n should be perceived as for INTG.
  for n=1:((l*sd)-(shift*sd))
   CURRENTWINDOW(n+(shift*sd))=INTG(n);
   end;

  CURRENTWINDOW=2*pi*h*SRCHTAB(row, col)*CURRENTWINDOW;

  SUMWINDOW=SUMWINDOW+CURRENTWINDOW;

  shift=shift+1;
  end;
```

```
% Plots to check the integrity of SUMWINDOWS.
% plot(SUMWINDOW);
% figure;

SUMWINDOW=SUMWINDOW+theta; % Add the initial phase.

% Taking the phase function for time period of the last symbol.
PHASEFUNCTION=[];
PHASEFUNCTION=SUMWINDOW(1, ((l-1)*sd+1):l*sd);
ROM(row, 1)=row; % Inserting the row number into the 1st col.

% Adding the phase function into the ROM table.
ROM(row, 2:(sd+1))=PHASEFUNCTION;
end;
```

rquad.m

```
% FILE: rquad.m
% ============
% This block stores the received quadrature signal into RQUAD.

function y=rquad(x)

% Side-effect variable.
global RQUAD;

RQUAD=[RQUAD,x];
y=x;
```

rsymbols.m

```
% FILE: rsymbols.m
% ==============
% This block records the received symbols into RSYMBOLS.

function y=rsymbols(x)

% side effect variable.
global RSYMBOLS;

RSYMBOLS=[RSYMBOLS,x];

y=x;
```

simac.m

```
% FILE: simac.m
% ==========
% SIMULINK file for Simulator 2: BER performance of CPM schemes in an AWGN
channel.

function [ret,x0,str,ts,xts]=simac(t,x,u,flag);
%SIMAC      is the M-file description of the SIMULINK system named SIMAC.
%       The block-diagram can be displayed by typing: SIMAC.
%
%       SYS=SIMAC(T,X,U,FLAG) returns depending on FLAG certain
%       system values given time point, T, current state vector, X,
%       and input vector, U.
%       FLAG is used to indicate the type of output to be returned in SYS.
%
%       Setting FLAG=1 causes SIMAC to return state derivatives, FLAG=2
%       discrete states, FLAG=3 system outputs and FLAG=4 next sample
%       time. For more information and other options see SFUNC.
%
%       Calling SIMAC with a FLAG of zero:
%       [SIZES]=SIMAC([],[],[],0), returns a vector, SIZES, which
%       contains the sizes of the state vector and other parameters.
%               SIZES(1) number of states
%               SIZES(2) number of discrete states
%               SIZES(3) number of outputs
%               SIZES(4) number of inputs
%               SIZES(5) number of roots (currently unsupported)
%               SIZES(6) direct feedthrough flag
%               SIZES(7) number of sample times
%
%       For the definition of other parameters in SIZES, see SFUNC.
%       See also, TRIM, LINMOD, LINSIM, EULER, RK23, RK45, ADAMS, GEAR.

% Note: This M-file is only used for saving graphical information;
%       after the model is loaded into memory an internal model
%       representation is used.

% the system will take on the name of this mfile:
sys = mfilename;
new_system(sys)
simver(1.3)
if (0 == (nargin + nargout))
   set_param(sys,'Location',[124,222,908,768])
   open_system(sys)
end;
set_param(sys,'algorithm',    'RK-45')
set_param(sys,'Start time',   '0.0')
```

```
set_param(sys,'Stop time',    '999999')
set_param(sys,'Min step size', 'ps')
set_param(sys,'Max step size', 'ps')
set_param(sys,'Relative error','1e-3')
set_param(sys,'Return vars',   '')

add_block('built-in/MATLAB Fcn',[sys,'/','Phase Pulse Generator'])
set_param([sys,'/','Phase Pulse Generator'],...
                'MATLAB Fcn','phasepg',....
                'position',[295,133,355,167])

add_block('built-in/Product',[sys,'/','Product1'])
set_param([sys,'/','Product1'],...
                'position',[830,163,860,187])

add_block('built-in/Product',[sys,'/','Product2'])
set_param([sys,'/','Product2'],...
                'position',[830,118,860,142])

add_block('built-in/MATLAB Fcn',[sys,'/','cos_'])
set_param([sys,'/','cos_'],...
                'MATLAB Fcn','cos',...
                'position',[825,43,885,77])

add_block('built-in/Mux',[sys,'/','Mux'])
set_param([sys,'/','Mux'],...
                'inputs','2',...
                'position',[1080,126,1110,159])

add_block('built-in/MATLAB Fcn',[sys,'/','Correlator'])
set_param([sys,'/','Correlator'],...
                'MATLAB Fcn','correl',...
                'position',[1145,128,1205,162])

add_block('built-in/Filter',[sys,'/','Filter1'])
set_param([sys,'/','Filter1'],...
                'Numerator','B',...
                'Denominator','A',...
                'Sample time','ps',...
                'position',[930,91,990,129])

add_block('built-in/Filter',[sys,'/','Filter'])
set_param([sys,'/','Filter'],...
                'Numerator','B',...
                'Denominator','A',...
                'Sample time','ps',...
                'position',[935,191,995,229])
```

```
add_block('built-in/MATLAB Fcn',[sys,'/','sin_'])
set_param([sys,'/','sin_'],...
                'position',[760,238,820,272])

add_block('built-in/Gain',[sys,'/','Gain1'])
set_param([sys,'/','Gain1'],...
                'Gain','2',...
                'position',[995,22,1020,48])

add_block('built-in/Gain',[sys,'/','Gain'])
set_param([sys,'/','Gain'],...
                'Gain','2',...
                'position',[1010,247,1035,273])

add_block('built-in/MATLAB Fcn',[sys,'/','Viterbi Detector'])
set_param([sys,'/','Viterbi Detector'],...
                'MATLAB Fcn','viterbi',...
                'position',[1430,128,1490,162])

add_block('built-in/MATLAB Fcn',[sys,'/','SYMBOLS'])
set_param([sys,'/','SYMBOLS'],...
                'MATLAB Fcn','symbols',...
                'position',[260,78,320,112])

add_block('built-in/MATLAB Fcn',[sys,'/','Random Symbol Generator'])
set_param([sys,'/','Random Symbol Generator'],...
                'MATLAB Fcn','randsym',...
                'position',[130,128,190,162])

add_block('built-in/Step Fcn',[sys,'/','Step Initiator'])
set_param([sys,'/','Step Initiator'],...
                'Time','0',...
                'Before','1',...
                'position',[30,135,50,155])


%    Subsystem  ['Discrete-Time',13,'Integrator'].

new_system([sys,'/',['Discrete-Time',13,'Integrator']])
set_param([sys,'/',['Discrete-Time',13,'Integrator']],'Location',[392,641,765,859])

add_block('built-in/Inport',[sys,'/',['Discrete-Time',13,'Integrator/in_1']])
set_param([sys,'/',['Discrete-Time',13,'Integrator/in_1']],...
                'position',[25,70,45,90])

add_block('built-in/Sum',[sys,'/',['Discrete-Time',13,'Integrator/Sum']])
set_param([sys,'/',['Discrete-Time',13,'Integrator/Sum']],...
                'position',[150,63,180,127])
```

```
add_block('built-in/Unit Delay',[sys,'/',['Discrete-Time',13,'Integrator/Unit Delay']])
set_param([sys,'/',['Discrete-Time',13,'Integrator/Unit Delay']],...
                'Sample time','Ts',...
                'x0','X0',...
                'position',[210,85,260,105])

add_block('built-in/Gain',[sys,'/',['Discrete-Time',13,'Integrator/Gain']])
set_param([sys,'/',['Discrete-Time',13,'Integrator/Gain']],...
                'Gain','Ts',...
                'position',[80,61,110,99])

add_block('built-in/Outport',[sys,'/',['Discrete-Time',13,'Integrator/out_1']])
set_param([sys,'/',['Discrete-Time',13,'Integrator/out_1']],...
                'position',[320,85,340,105])
add_line([sys,'/',['Discrete-Time',13,'Integrator']],[115,80;145,80])
add_line([sys,'/',['Discrete-Time',13,'Integrator']],[50,80;75,80])
add_line([sys,'/',['Discrete-Time',13,'Integrator']],[185,95;205,95])
add_line([sys,'/',['Discrete-Time',13,'Integrator']],[265,95;315,95])
add_line([sys,'/',['Discrete-
Time',13,'Integrator']],[285,95;285,165;120,165;120,110;145,110])
set_param([sys,'/',['Discrete-Time',13,'Integrator']],...
                'Mask Display','dpoly(Ts,[1 -1],"z")',...
                'Mask Type','Discrete-Time Integrator',...
                'Mask Dialogue','Discrete-Time Integrator:|Initial Condition:|Sample
Time:')
set_param([sys,'/',['Discrete-Time',13,'Integrator']],...
                'Mask Translate','X0=@1;Ts=@2;',...
                'Mask Help','Implements a zeroth order discrete\nintegration using a gain,
a sum, and\na unit delay. Inputs may be scalar\nor vector.\n')
set_param([sys,'/',['Discrete-Time',13,'Integrator']],...
                'Mask Entries','0\/psV')


%    Finished composite block ['Discrete-Time',13,'Integrator'].

set_param([sys,'/',['Discrete-Time',13,'Integrator']],...
                'position',[375,129,415,171])

add_block('built-in/MATLAB Fcn',[sys,'/','RINPHASE'])
set_param([sys,'/','RINPHASE'],...
                'MATLAB Fcn','rinphase',...
                'position',[1035,58,1095,92])

add_block('built-in/MATLAB Fcn',[sys,'/','RQUAD'])
set_param([sys,'/','RQUAD'],...
                'MATLAB Fcn','rquad',...
                'position',[1050,203,1110,237])
```

```
add_block('built-in/To Workspace',[sys,'/','To Workspace2'])
set_param([sys,'/','To Workspace2'],...
                'mat-name','RINPHASEUF',...
                'position',[900,7,950,23])

add_block('built-in/To Workspace',[sys,'/','To Workspace3'])
set_param([sys,'/','To Workspace3'],...
                'mat-name','RQUADUF',...
                'position',[890,257,940,273])

add_block('built-in/Sum',[sys,'/','Sum1'])
set_param([sys,'/','Sum1'],...
                'position',[625,155,645,175])

add_block('built-in/Demux',[sys,'/','Demux'])
set_param([sys,'/','Demux'],...
                'outputs','2',...
                'position',[1250,126,1290,159])

add_block('built-in/Sum',[sys,'/','Sum'])
set_param([sys,'/','Sum'],...
                'position',[1360,135,1380,155])

add_block('built-in/MATLAB Fcn',[sys,'/','RSYMBOLS'])
set_param([sys,'/','RSYMBOLS'],...
                'MATLAB Fcn','rsymbols',...
                'position',[1515,128,1575,162])

add_block('built-in/To Workspace',[sys,'/','To Workspace4'])
set_param([sys,'/','To Workspace4'],...
                'mat-name','AWGN',...
                'position',[625,22,675,38])

add_block('built-in/MATLAB Fcn',[sys,'/',['Symbol Comparator and ',13,'Simulation
Regulator']])
set_param([sys,'/',['Symbol Comparator and ',13,'Simulation Regulator']],...
                'MATLAB Fcn','compreg',...
                'position',[1635,128,1695,162])

add_block('built-in/Stop Simulation',[sys,'/','Stop Simulation'])
set_param([sys,'/','Stop Simulation'],...
                'position',[1780,128,1820,162])

add_block('built-in/To Workspace',[sys,'/','To Workspace1'])
set_param([sys,'/','To Workspace1'],...
                'mat-name','CPM',...
                'position',[660,82,710,98])
```

```
add_block('built-in/MATLAB Fcn',[sys,'/','Noise Regulator'])
set_param([sys,'/','Noise Regulator'],...
                'MATLAB Fcn','noisereg',...
                'position',[530,53,590,87])

add_block('built-in/Gain',[sys,'/','Gain2'])
set_param([sys,'/','Gain2'],...
                'Gain','wc*ps',...
                'position',[250,203,280,247])

add_block('built-in/Sum',[sys,'/','Sum2'])
set_param([sys,'/','Sum2'],...
                'position',[340,220,360,240])

add_block('built-in/Unit Delay',[sys,'/','Unit Delay'])
set_param([sys,'/','Unit Delay'],...
                'Sample time','ps',...
                'position',[425,222,475,238])

add_block('built-in/Sum',[sys,'/','Sum3'])
set_param([sys,'/','Sum3'],...
                'position',[495,140,515,160])

add_block('built-in/MATLAB Fcn',[sys,'/','PHASEWAVEFORM'])
set_param([sys,'/','PHASEWAVEFORM'],...
                'MATLAB Fcn','phasewf',...
                'position',[440,88,500,122])

add_block('built-in/Gain',[sys,'/','Gain3'])
set_param([sys,'/','Gain3'],...
                'Gain','a',...
                'position',[580,168,610,212])

add_block('built-in/MATLAB Fcn',[sys,'/','cos'])
set_param([sys,'/','cos'],...
                'MATLAB Fcn','cos',...
                'position',[505,183,565,217])

add_block('built-in/Gain',[sys,'/','Gain4'])
set_param([sys,'/','Gain4'],...
                'Gain','-1',...
                'position',[695,233,725,277])

add_block('built-in/To Workspace',[sys,'/','To Workspace5'])
set_param([sys,'/','To Workspace5'],...
                'mat-name','NOISELESSCPM',...
                'buffer','2500',...
```

```
                'position',[590,283,680,297])

add_block('built-in/White Noise',[sys,'/',['Random',13,'Number']])
set_param([sys,'/',['Random',13,'Number']],...
                'position',[465,12,510,48])
add_line(sys,[360,150;370,150])
add_line(sys,[55,145;125,145])
add_line(sys,[890,60;895,60;895,95;810,95;810,125;825,125])
add_line(sys,[865,175;905,175;905,210;930,210])
add_line(sys,[1115,145;1140,145])
add_line(sys,[1210,145;1245,145])
add_line(sys,[1295,135;1355,140])
add_line(sys,[1295,150;1355,150])
add_line(sys,[1385,145;1425,145])
add_line(sys,[865,130;890,130;890,110;925,110])
add_line(sys,[825,255;835,255;835,225;810,225;810,180;825,180])
add_line(sys,[995,110;1005,110;1005,80;975,80;975,35;990,35])
add_line(sys,[1000,210;1010,210;1010,235;990,235;990,260;1005,260])
add_line(sys,[195,145;220,145;255,95])
add_line(sys,[325,95;335,95;335,120;275,120;275,150;290,150])
add_line(sys,[420,150;435,105])
add_line(sys,[1025,35;1035,35;1035,50;1025,50;1025,65;1015,65;1015,75;1030,75])
add_line(sys,[1100,75;1110,75;1110,105;1060,105;1060,135;1075,135])
add_line(sys,[1040,260;1050,260;1050,250;1040,250;1040,235;1030,235;1030,220;104
5,220])
add_line(sys,[1115,220;1125,220;1125,195;1060,195;1060,150;1075,150])
add_line(sys,[865,130;885,130;895,15])
add_line(sys,[865,175;870,175;870,265;885,265])
add_line(sys,[1495,145;1510,145])
add_line(sys,[1580,145;1630,145])
add_line(sys,[650,165;730,165;730,135;825,135])
add_line(sys,[650,165;730,165;730,170;825,170])
add_line(sys,[650,165;660,165;660,140;625,140;625,90;655,90])
add_line(sys,[1700,145;1775,145])
add_line(sys,[515,30;525,70])
add_line(sys,[595,70;605,70;605,160;620,160])
add_line(sys,[285,225;335,225])
add_line(sys,[365,230;420,230])
add_line(sys,[480,230;495,230;490,255;320,255;320,235;335,235])
add_line(sys,[505,105;515,105;515,130;475,130;470,145;490,145])
add_line(sys,[480,230;490,155])
add_line(sys,[55,145;80,145;80,165;145,165;145,225;245,225])
add_line(sys,[520,150;530,150;530,165;485,165;485,200;500,200])
add_line(sys,[570,200;575,190])
add_line(sys,[615,190;620,170])
add_line(sys,[480,230;645,230;645,210;650,210;650,60;820,60])
add_line(sys,[480,230;580,230;580,255;690,255])
add_line(sys,[730,255;755,255])
```

```
add_line(sys,[615,190;625,190;625,235;570,235;570,290;585,290])
add_line(sys,[595,70;600,70;600,30;620,30])

drawnow

% Return any arguments.
if (nargin | nargout)
        % Must use feval here to access system in memory
        if (nargin > 3)
                if (flag == 0)
                        eval(['[ret,x0,str,ts,xts]=',sys,'(t,x,u,flag);'])
                else
                        eval(['ret =', sys,'(t,x,u,flag);'])
                end
        else
                [ret,x0,str,ts,xts] = feval(sys);
        end
else
        drawnow % Flash up the model and execute load callback
end
```

simrc.m

```
% FILE: simrc.m
% ============
% SIMULINK file for Simulator 3: BER performance of CPM schemes in a
% multipath channel (Rician & AWGN channel).

function [ret,x0,str,ts,xts]=simrc(t,x,u,flag);
%SIMRC      is the M-file description of the SIMULINK system named SIMRC.
%       The block-diagram can be displayed by typing: SIMRC.
%
%       SYS=SIMRC(T,X,U,FLAG) returns depending on FLAG certain
%       system values given time point, T, current state vector, X,
%       and input vector, U.
%       FLAG is used to indicate the type of output to be returned in SYS.
%
%       Setting FLAG=1 causes SIMRC to return state derivatives, FLAG=2
%       discrete states, FLAG=3 system outputs and FLAG=4 next sample
%       time. For more information and other options see SFUNC.
%
%       Calling SIMRC with a FLAG of zero:
%       [SIZES]=SIMRC([],[],[],0), returns a vector, SIZES, which
%       contains the sizes of the state vector and other parameters.
%               SIZES(1) number of states
%               SIZES(2) number of discrete states
```

```
%               SIZES(3) number of outputs
%               SIZES(4) number of inputs
%               SIZES(5) number of roots (currently unsupported)
%               SIZES(6) direct feedthrough flag
%               SIZES(7) number of sample times
%
%       For the definition of other parameters in SIZES, see SFUNC.
%       See also, TRIM, LINMOD, LINSIM, EULER, RK23, RK45, ADAMS, GEAR.

% Note: This M-file is only used for saving graphical information;
%       after the model is loaded into memory an internal model
%       representation is used.

% the system will take on the name of this mfile:
sys = mfilename;
new_system(sys)
simver(1.3)
if (0 == (nargin + nargout))
   set_param(sys,'Location',[74,152,1110,624])
   open_system(sys)
end;
set_param(sys,'algorithm',     'RK-45')
set_param(sys,'Start time',    '0.0')
set_param(sys,'Stop time',     '999999')
set_param(sys,'Min step size', 'ps')
set_param(sys,'Max step size', 'ps')
set_param(sys,'Relative error','1e-3')
set_param(sys,'Return vars',   '')

add_block('built-in/MATLAB Fcn',[sys,'/','Phase Pulse Generator'])
set_param([sys,'/','Phase Pulse Generator'],...
                'MATLAB Fcn','phasepg',...
                'position',[295,133,355,167])

add_block('built-in/Product',[sys,'/','Product1'])
set_param([sys,'/','Product1'],...
                'position',[830,163,860,187])

add_block('built-in/Product',[sys,'/','Product'])
set_param([sys,'/','Product'],...
                'position',[830,118,860,142])

add_block('built-in/Mux',[sys,'/','Mux'])
set_param([sys,'/','Mux'],...
                'inputs','2',...
                'position',[1080,126,1110,159])

add_block('built-in/MATLAB Fcn',[sys,'/','Correlator'])
```

```
set_param([sys,'/','Correlator'],...
                'MATLAB Fcn','correl',...
                'position',[1145,128,1205,162])

add_block('built-in/Filter',[sys,'/','Filter1'])
set_param([sys,'/','Filter1'],...
                'Numerator','B',...
                'Denominator','A',...
                'Sample time','ps',...
                'position',[930,91,990,129])

add_block('built-in/Filter',[sys,'/','Filter'])
set_param([sys,'/','Filter'],...
                'Numerator','B',...
                'Denominator','A',...
                'Sample time','ps',...
                'position',[935,191,995,229])

add_block('built-in/Gain',[sys,'/','Gain1'])
set_param([sys,'/','Gain1'],...
                'Gain','2',...
                'position',[995,22,1020,48])

add_block('built-in/Gain',[sys,'/','Gain'])
set_param([sys,'/','Gain'],...
                'Gain','2',...
                'position',[1010,247,1035,273])

add_block('built-in/MATLAB Fcn',[sys,'/','Viterbi Detector'])
set_param([sys,'/','Viterbi Detector'],...
                'MATLAB Fcn','viterbi',...
                'position',[1430,128,1490,162])

add_block('built-in/MATLAB Fcn',[sys,'/','SYMBOLS'])
set_param([sys,'/','SYMBOLS'],...
                'MATLAB Fcn','symbols',...
                'position',[260,78,320,112])

add_block('built-in/MATLAB Fcn',[sys,'/','Random Symbol Generator'])
set_param([sys,'/','Random Symbol Generator'],...
                'MATLAB Fcn','randsym',...
                'position',[130,128,190,162])

add_block('built-in/Step Fcn',[sys,'/','Step Initiator'])
set_param([sys,'/','Step Initiator'],...
                'Time','0',...
                'Before','1',...
                'position',[30,135,50,155])
```

```
add_block('built-in/MATLAB Fcn',[sys,'/','RINPHASE'])
set_param([sys,'/','RINPHASE'],...
                'MATLAB Fcn','rinphase',...
                'position',[1035,58,1095,92])

add_block('built-in/MATLAB Fcn',[sys,'/','RQUAD'])
set_param([sys,'/','RQUAD'],...
                'MATLAB Fcn','rquad',...
                'position',[1050,203,1110,237])

add_block('built-in/To Workspace',[sys,'/','To Workspace2'])
set_param([sys,'/','To Workspace2'],...
                'mat-name','RINPHASEUF',...
                'position',[900,7,950,23])

add_block('built-in/To Workspace',[sys,'/','To Workspace3'])
set_param([sys,'/','To Workspace3'],...
                'mat-name','RQUADUF',...
                'position',[890,257,940,273])

add_block('built-in/Demux',[sys,'/','Demux'])
set_param([sys,'/','Demux'],...
                'outputs','2',...
                'position',[1250,126,1290,159])

add_block('built-in/Sum',[sys,'/','Sum'])
set_param([sys,'/','Sum'],...
                'position',[1360,135,1380,155])

add_block('built-in/MATLAB Fcn',[sys,'/','RSYMBOLS'])
set_param([sys,'/','RSYMBOLS'],...
                'MATLAB Fcn','rsymbols',...
                'position',[1515,128,1575,162])

add_block('built-in/MATLAB Fcn',[sys,'/',['Symbol Comparator and ',13,'Simulation
Regulator']])
set_param([sys,'/',['Symbol Comparator and ',13,'Simulation Regulator']],...
                'MATLAB Fcn','compreg',...
                'position',[1635,128,1695,162])

add_block('built-in/Stop Simulation',[sys,'/','Stop Simulation'])
set_param([sys,'/','Stop Simulation'],...
                'position',[1780,128,1820,162])

add_block('built-in/MATLAB Fcn',[sys,'/','PHASEWAVEFORM'])
set_param([sys,'/','PHASEWAVEFORM'],...
                'MATLAB Fcn','phasewf',...
```

```
                'position',[450,133,510,167])


%    Subsystem  ['Discrete-Time',13,'Integrator'].

new_system([sys,'/',['Discrete-Time',13,'Integrator']])
set_param([sys,'/',['Discrete-Time',13,'Integrator']],'Location',[392,641,765,859])

add_block('built-in/Inport',[sys,'/',['Discrete-Time',13,'Integrator/in_1']])
set_param([sys,'/',['Discrete-Time',13,'Integrator/in_1']],...
                'position',[25,70,45,90])

add_block('built-in/Sum',[sys,'/',['Discrete-Time',13,'Integrator/Sum']])
set_param([sys,'/',['Discrete-Time',13,'Integrator/Sum']],...
                'position',[150,63,180,127])

add_block('built-in/Unit Delay',[sys,'/',['Discrete-Time',13,'Integrator/Unit Delay']])
set_param([sys,'/',['Discrete-Time',13,'Integrator/Unit Delay']],...
                'Sample time','Ts',...
                'x0','X0',...
                'position',[210,85,260,105])

add_block('built-in/Gain',[sys,'/',['Discrete-Time',13,'Integrator/Gain']])
set_param([sys,'/',['Discrete-Time',13,'Integrator/Gain']],...
                'Gain',"Ts',...
                'position',[80,61,110,99])

add_block('built-in/Outport',[sys,'/',['Discrete-Time',13,'Integrator/out_1']])
set_param([sys,'/',['Discrete-Time',13,'Integrator/out_1']],...
                'position',[320,85,340,105])
add_line([sys,'/',['Discrete-Time',13,'Integrator']],[115,80;145,80])
add_line([sys,'/',['Discrete-Time',13,'Integrator']],[50,80;75,80])
add_line([sys,'/',['Discrete-Time',13,'Integrator']],[185,95;205,95])
add_line([sys,'/',['Discrete-Time',13,'Integrator']],[265,95;315,95])
add_line([sys,'/',['Discrete-
Time',13,'Integrator']],[285,95;285,165;120,165;120,110;145,110])
set_param([sys,'/',['Discrete-Time',13,'Integrator']],...
                'Mask Display','dpoly(Ts,[1 -1],"z")',...
                'Mask Type','Discrete-Time Integrator',...
                'Mask  Dialogue','Discrete-Time  Integrator:|Initial  Condition:|Sample
Time:')
set_param([sys,'/',['Discrete-Time',13,'Integrator']],...
                'Mask Translate','X0=@1;Ts=@2;',...
                'Mask Help','Implements a zeroth order discrete\nintegration using a gain,
a sum, and\na unit delay. Inputs may be scalar\nor vector.\n')
set_param([sys,'/',['Discrete-Time',13,'Integrator']],...
                'Mask Entries','0\/ps\/')
```

```
%    Finished composite block ['Discrete-Time',13,'Integrator'].

set_param([sys,'/',['Discrete-Time',13,'Integrator']],...
                'position',[380,129,420,171])

add_block('built-in/Gain',[sys,'/','Gain2'])
set_param([sys,'/','Gain2'],...
                'Gain','wc*ps',...
                'position',[305,223,335,267])

add_block('built-in/Sum',[sys,'/','Sum2'])
set_param([sys,'/','Sum2'],...
                'position',[395,240,415,260])

add_block('built-in/Unit Delay',[sys,'/','Unit Delay'])
set_param([sys,'/','Unit Delay'],...
                'Sample time','ps',...
                'position',[480,242,530,258])

add_block('built-in/Sum',[sys,'/','Sum3'])
set_param([sys,'/','Sum3'],...
                'position',[530,145,550,165])

add_block('built-in/MATLAB Fcn',[sys,'/','sin_'])
set_param([sys,'/','sin_'],...
                'position',[750,233,810,267])

add_block('built-in/MATLAB Fcn',[sys,'/','cos_'])
set_param([sys,'/','cos_'],...
                'MATLAB Fcn','cos',...
                'position',[765,43,825,77])

add_block('built-in/MATLAB Fcn',[sys,'/','sin'])
set_param([sys,'/','sin'],...
                'position',[570,173,630,207])

add_block('built-in/To Workspace',[sys,'/','To Workspace1'])
set_param([sys,'/','To Workspace1'],...
                'mat-name','CPM',...
                'position',[730,92,780,108])

add_block('built-in/MATLAB Fcn',[sys,'/','cos'])
set_param([sys,'/','cos'],...
                'MATLAB Fcn','cos',...
                'position',[570,113,630,147])

add_block('built-in/Product',[sys,'/','Product2'])
```

```
set_param([sys,'/','Product2'],...
                'position',[655,113,685,137])

add_block('built-in/Product',[sys,'/','Product3'])
set_param([sys,'/','Product3'],...
                'position',[650,183,680,207])

add_block('built-in/Sum',[sys,'/','Sum4'])
set_param([sys,'/','Sum4'],...
                'inputs','+++',...
                'position',[730,152,750,188])

add_block('built-in/MATLAB Fcn',[sys,'/','Noise Regulator 1'])
set_param([sys,'/','Noise Regulator 1'],...
                'MATLAB Fcn','rcreg',...
                'position',[525,13,585,47])

add_block('built-in/MATLAB Fcn',[sys,'/','Noise Regulator 2'])
set_param([sys,'/','Noise Regulator 2'],...
                'MATLAB Fcn','rcreg',...
                'position',[540,308,600,342])

add_block('built-in/Sum',[sys,'/','Sum5'])
set_param([sys,'/','Sum5'],...
                'position',[625,25,645,45])

add_block('built-in/Gain',[sys,'/','Gain3'])
set_param([sys,'/','Gain3'],...
                'Gain','a',...
                'position',[495,68,525,112])

add_block('built-in/Gain',[sys,'/','Gain4'])
set_param([sys,'/','Gain4'],...
                'Gain','-1',...
                'position',[680,237,705,263])

add_block('built-in/To Workspace',[sys,'/','To Workspace'])
set_param([sys,'/','To Workspace'],...
                'mat-name','AWGN',...
                'position',[690,7,740,23])

add_block('built-in/White Noise',[sys,'/',['Random',13,'Number']])
set_param([sys,'/',['Random',13,'Number']],...
                'Seed','0.1',...
                'position',[430,12,475,48])

add_block('built-in/White Noise',[sys,'/',['Random',13,'Number1']])
set_param([sys,'/',['Random',13,'Number1']],...
```

```
                'position',[445,307,490,343])

add_block('built-in/White Noise',[sys,'/',['Random',13,'Number2']])
set_param([sys,'/',['Random',13,'Number2']],...
                'Seed','0.2',...
                'position',[445,377,490,413])

add_block('built-in/MATLAB Fcn',[sys,'/','Noise Regulator 3'])
set_param([sys,'/','Noise Regulator 3'],...
                'MATLAB Fcn','noisereg',...
                'position',[540,378,600,412])
add_line(sys,[360,150;375,150])
add_line(sys,[55,145;125,145])
add_line(sys,[830,60;895,60;895,95;810,95;810,125;825,125])
add_line(sys,[865,175;905,175;905,210;930,210])
add_line(sys,[1115,145;1140,145])
add_line(sys,[1210,145;1245,145])
add_line(sys,[1295,135;1355,140])
add_line(sys,[1295,150;1355,150])
add_line(sys,[1385,145;1425,145])
add_line(sys,[865,130;890,130;890,110;925,110])
add_line(sys,[815,250;835,250;835,225;810,225;810,180;825,180])
add_line(sys,[995,110;1005,110;1005,80;975,80;975,35;990,35])
add_line(sys,[1000,210;1010,210;1010,235;990,235;990,260;1005,260])
add_line(sys,[195,145;220,145;255,95])
add_line(sys,[325,95;335,95;335,120;275,120;275,150;290,150])
add_line(sys,[425,150;445,150])
add_line(sys,[1025,35;1035,35;1035,50;1025,50;1025,65;1015,65;1015,75;1030,75])
add_line(sys,[1100,75;1110,75;1110,105;1060,105;1060,135;1075,135])
add_line(sys,[1040,260;1050,260;1050,250;1040,250;1040,235;1030,235;1030,220;104
5,220])
add_line(sys,[1115,220;1125,220;1125,195;1060,195;1060,150;1075,150])
add_line(sys,[865,130;885,130;895,15])
add_line(sys,[865,175;870,175;870,265;885,265])
add_line(sys,[1495,145;1510,145])
add_line(sys,[1580,145;1630,145])
add_line(sys,[1700,145;1775,145])
add_line(sys,[480,30;520,30])
add_line(sys,[340,245;390,245])
add_line(sys,[420,250;475,250])
add_line(sys,[55,145;80,145;80,165;245,165;245,245;300,245])
add_line(sys,[535,250;545,250;545,275;375,275;375,255;390,255])
add_line(sys,[515,150;525,150])
add_line(sys,[535,250;545,250;545,210;515,210;525,160])
add_line(sys,[535,250;615,250;615,60;760,60])
add_line(sys,[555,155;565,130])
add_line(sys,[555,155;565,190])
add_line(sys,[635,130;650,130])
```

```
add_line(sys,[635,190;645,190])
add_line(sys,[690,125;700,125;700,160;725,160])
add_line(sys,[685,195;700,195;700,170;725,170])
add_line(sys,[755,170;785,170;785,135;825,135])
add_line(sys,[755,170;825,170])
add_line(sys,[755,170;765,170;765,140;710,140;710,100;725,100])
add_line(sys,[495,325;535,325])
add_line(sys,[605,325;635,325;645,200])
add_line(sys,[590,30;620,30])
add_line(sys,[650,35;660,35;660,65;635,65;635,120;650,120])
add_line(sys,[530,90;590,90;590,40;620,40])
add_line(sys,[55,145;95,145;95,120;425,120;425,90;490,90])
add_line(sys,[535,250;675,250])
add_line(sys,[710,250;745,250])
add_line(sys,[590,30;600,30;600,15;685,15])
add_line(sys,[495,395;535,395])
add_line(sys,[605,395;710,395;710,180;725,180])

drawnow

% Return any arguments.
if (nargin | nargout)
        % Must use feval here to access system in memory
        if (nargin > 3)
                if (flag == 0)
                        eval(['[ret,x0,str,ts,xts]=',sys,'(t,x,u,flag);'])
                else
                        eval(['ret =', sys,'(t,x,u,flag);'])
                end
        else
                [ret,x0,str,ts,xts] = feval(sys);
        end
else
        drawnow % Flash up the model and execute load callback
end
```

srchtab.m

```
% FILE: srchtab.m
% ==============
% This function generates the search table.

function srchtab()

% 1st col is possible phases. Last col is search key.
% The rest are symbol combinations.
```

```
% Size: pml x (l+2)
% Denoted: pp x qq

% For debugging purposes:
% POSSPHAS = vector of all possible phases.
% m = the number of different symbols.
% SYMMAT = vector of all symbols.
% l = L. The duration of the phase function expressed as the number of T's.

% Input variable.
% Symbol matrix.
global SYMMAT;
% Possible phase vector, signalling levels, length of phase pulse.
global POSSPHAS m l;

% Side-effect variable.
global SRCHTAB;

qq=l+2;
% Adding the symbol combinations into srchtab.
sizePOSSPHAS=size(POSSPHAS, 2);
sizeSYMMAT=size(SYMMAT, 2);
pp=sizePOSSPHAS*(m^l);

% Symbol combinations start from the second last col.
q=qq-1;

% Generating symbol combination.
while q>=2
% sc = symbol count, hsc = hold symbol count, pc = phase count, hpc = hold phase
count.
 sc=1;
 hsc=1;
 symbol=SYMMAT(sc);
 p=1;
 while p<=pp
  SEARCHTB(q,p)=symbol;
  if hsc==m^((abs(q-qq)+1)-2)
   hsc=0;
   sc=sc+1;
   if sc>sizeSYMMAT
    sc=1;
    end;
   symbol=SYMMAT(sc);
   end;
  p=p+1;
  hsc=hsc+1;
  end;
```

```
 q=q-1;
 end;

% Generating the phase column.
phase=POSSPHAS(1);
p=1;
pc=1;
hpc=1;
while p<=pp
 SEARCHTB(1,p)=phase;
 if hpc==m^l
  hpc=0;
  pc=pc+1;
  if pc>sizePOSSPHAS
   pc=1;
   end;
  phase=POSSPHAS(pc);
  end;
 p=p+1;
 hpc=hpc+1;
 end;

SEARCHTB=SEARCHTB';

% Generating index.
for n=1:pp
 INDEX(n)=n;
 end;
INDEX=INDEX';

SEARCHTB(:, [qq])=INDEX(:, [1]);

SRCHTAB=SEARCHTB;
```

statetab.m

```
% FILE: statetab.m
% ================
% This function generates the backward lookup table for the demodulator.
% It shows for a particular state, all states which the current state could
% have originated from.

% This function generates STATETAB.
% STATETAB is a table showing all the possible states in terms of their initial
% phase and l-1 symbol combinations in order.
% The first column of the table is an index of the states.
```

```
% The second column of the table is the phase state and subsequent columns hold
% correlative state combinations.
% For full response schemes, only the phase state and state index columns are present
% in the table.
% STATETAB is pm^lx(l-1).

function statetab()

% Input variables.
global POSSPHAS m l h;

% Side-effect variables.
global STATETAB;

% For debugging purposes:
% global POSSPHAS m l;
% POSSPHAS=possphas(1,3);
% m=8;
% l=1;
% h=1/3;

% Threshold for error.
error=0.001;

% Constructing the index and phase columns of STATETAB.
sizePOSSPHAS=size(POSSPHAS,2);
mlmin1=m^(l-1);
count3=1;
for count1=1:sizePOSSPHAS
 for count2=1:mlmin1
  INDEXSLICE(count3,1)=count3;
  PHASESLICE(count3,1)=POSSPHAS(count1);
  count3=count3+1;
  end;
 end;

% Generating the symbol matrix.
SYMMAT=[];
n=0;
while (2*n+1)<=(m-1)
 SYMMAT=[-(2*n+1), SYMMAT, (2*n+1)];
 n=n+1;
end;

% The STATETAB's size is denoted as pp x qq.
qq=l-1+2;

% Adding the symbol combinations into searchtab.
```

```
sizeSYMMAT=size(SYMMAT, 2);
pp=sizePOSSPHAS*(m^(l-1));

% Symbol combinations start from the last col.
q=qq;
while q>=3
% sc = symbol count, hsc = hold symbol count, pc = phase count, hpc = hold phase
count.
 sc=1;
 hsc=1;
 symbol=SYMMAT(sc);
 p=1;
 while p<=pp
  STATETAB(p,q)=symbol;
  if hsc==m^(abs(q-qq))
   hsc=0;
   sc=sc+1;
   if sc>sizeSYMMAT
    sc=1;
    end;
   symbol=SYMMAT(sc);
   end;
  p=p+1;
  hsc=hsc+1;
  end;
 q=q-1;
 end;

% Adding the phase and index columns.
STATETAB(:,2)=PHASESLICE(:,1);
STATETAB(:,1)=INDEXSLICE(:,1);
```

statetta.m

```
% FILE: statetta.m
% ================
% This function generates the state transition table, STATETTAB.
% STATETTAB shows all the possible state transitions during
% each symbol interval.

function statettab()

% Input variables.
global STATETAB
global SRCHTAB l h;
```

```
% Side-effect variables.
global STATETTAB;

% For debugging purposes.
%Test Parameters.
%h=0.5;
%m=2;
%l=3;
%POSSPHAS=possphas(1,2);
%SRCHTAB=searchtab(POSSPHAS,m,l);
%bkwdlt(POSSPHAS,m,l,h);

% Error threshold.
error=0.001;

% The size of SRCHTAB is denoted as pp x qq.
[pp,qq]=size(SRCHTAB);

% The SRCHTAB holds all possible state transitions.
% So the idea is to convert each row of the SRCHTAB into two states,
% namely the "current" state and the "previous" state.
for count1=1:pp
 % Determining the previous and current phases.
 currentphase=rem(SRCHTAB(count1,1)+pi*h*SRCHTAB(count1,2),2*pi);
 % Converting negative phase to positive.
 currentphase=rem(currentphase+2*pi,2*pi);
 % If the phase is close to 2*pi, then it must be 2*pi.
 if (abs(currentphase-2*pi)<error) | (abs(currentphase-0)<error)
  currentphase=0;
  end;

 prevphase=SRCHTAB(count1,1);

 % Determining the previous and current l-1 symbols.
 if l>1
  CURRENTLMIN1SYM=SRCHTAB(count1,3:(qq-1));
  PREVLMIN1SYM=SRCHTAB(count1,2:(qq-2));
 elseif l==1
  CURRENTLMIN1SYM=[];
  PREVLMIN1SYM=[];
 end;

 % Searching for the current state on the STATETAB using the current
 % phase and the current l-1 symbols.
 count2=1;
 if l>1
                    while (abs(prevphase-STATETAB(count2,2))>error |
~all(PREVLMIN1SYM==STATETAB(count2,3:(2+l-1))) )
```

```
    count2=count2+1;
    end;
 elseif l==1
   while abs(prevphase-STATETAB(count2,2))>error
    count2=count2+1;
    end;
 end;
 prevstate=count2;
 STATETTAB(count1,1)=prevstate;

 % Searching for the previous state on the STATETAB using the previous
 % phase and the previous l-1 symbols.
 count3=1;
 if l>1
                          while      (abs(currentphase-STATETAB(count3,2))>error      |
~all(CURRENTLMIN1SYM==STATETAB(count3,3:(2+l-1))) )
    count3=count3+1;
    end;
 elseif l==1
   while abs(currentphase-STATETAB(count3,2))>error
    count3=count3+1;
    end;
 end;
 currentstate=count3;
 STATETTAB(count1,2)=currentstate;

 end;
```

symbols.m

```
% FILE: symbols.m
% ==============
% This block stores the symbol stream into SYMBOLS.

function y=symbols(x)

% Side-effect variable.
global SYMBOLS;

SYMBOLS=[SYMBOLS,x];
y=x;
```

viterbi.m

```
% FILE: viterbi.m
% ================
% Block description
% -----------------
% This block converts data from CCTAB into received symbols.
% The input to this block are timing signals from the CC block.
% The output of this block are received symbols.
% A record of the states traversed is kept in VSTATEPROGTAB.
% (Note that for l==1, VSTATEPROGTAB cannot provide the symbols transmitted.)

% This block uses the Viterbi algorithm for symbol detection.

% VSTATEPROGTAB starts at position l-1 because the initial state is
% determined by considering the first l symbols (the first transition).
% For l==1, VSTATEPROGTAB10 holds the first initial state of the first transition.
% The second initial state is stored in VSTATEPROGTAB(1,1).

% Nt
% --
% The Viterbi decoder remembers the previous Nt states.
% The delay of this block is also Nt.
% Nt for this decoder is different to the Nt of the ML decoder.

% Min Nt=1
% --------
% When Nt=0:
% No cum CC values are remembered for previous states.
% No survival paths exists.
% Only the transitions are compared for the ML.
% This is ML decoder with Nt=1.
% When Nt=1, cum CC values for the previous one states are still remembered.
% Survival paths of length one exists.
% The next state transition and the previous cum CC values of the
% survival paths are considered at each decoding instance.

% Min l=1
% -------
% When l=1 and rectangular pulse shaping is used, we get CPFSK.
% In all CPM schemes, l=1 is the minimum because all state transitions
% are still considered at each decoding instance. Unlike FSK and PSK which do
% not consider state transitions.

% Delay
% -----
% The first symbol to be output from this decoder is delayed by Nt and
% it is the l th symbol of the transmitted symbol stream.
```

```
% This symbol represents the second state.

% Special consideration for l==1
% ------------------------------
% In partial response CPM (l>1), state transitions are considered
% to determine the actual symbols generated.
% There are pml state combinations between any two symbol intervals.
% There are also pml signal transitions due to the pm^l-1 states and m
% possible symbols.
% Thus there is a direct map between state combinations and signal transitions.
% So, by considering the state combinations we can determine the
% corresponding signal transition to determine which symbol was transmitted.

% With full response CPM (l<1), the possible states are the the possible
% phases the modulation scheme can attain at the end of each symbol interval.
% For h=1/2=>p=4 m=8 l=1, there are only 4x4=16 possible state combinations
% compared with pxm=32 actual signal transitions.
% Thus we cannot map the actual signal transitions with state combinations.
% To determine which signal was transmitted we just cannot consider the
% state combinations since many signal transitions have the same state combinations.
% Signal transitions show exactly what happened during modulation. It is by
% knowing which signal transitions were taken that we are able to determine
% which symbols were sent.
% Knowing the state transitions that was taken will not provide the symbols
% received because for each state transition, many symbols could be possible
% with full response schemes.

% For full response, no searches should be performed on the STATETTAB to find
% matching state transitions.

function y=viterbi(x)

% Input variables.
global STATETAB STATETABpp STATETTAB CCTAB SYMBOLS SRCHTAB sd
nt l m mincumccval;

% Side-effect variables.
global SPCCTAB SPTAB SPSYMTAB VSTATEPROGTAB10 VSTATEPROGTAB
ACTUALPATHCCVAL vcnt vcurrentsymbol;
% FIRSTLSYMBOLS declared globally for debugging purposes.
global FIRSTLSYMBOLS;
global fid;

% Local variables.
% Initialise the extended surviving paths tables.
ESPCCTAB=[];
ESPTAB=[];
ESPSYMTAB=[];
```

```
% Initialise temporary storage of new survival paths.
TEMPSPTAB=[];
TEMPSPCCTAB=[];
TEMPSPSYMTAB=[];

% On the last time slice of a symbol interval.
if (x==1)
 vcnt=vcnt+1;

 % First l-1 symbols are skipped.

 if vcnt==l
  % Determining the initial two states of the first l symbols.
  % This is done by looking at what was actually sent.
  % This is suitable only in simulation to avoid considering all possible paths
  % as with the ml detector.
  % Getting the first l symbols.
  LPTS=[1:l]*sd;
  FIRSTLSYMBOLS=SYMBOLS(1,LPTS);
  count10=1;
  istatet=1;
  if l==1
   % If it's a full response scheme, we only look at the first symbol.
   while count10<=(m^l)
    % Just considering the states with 0 initial phases.
    if all(SRCHTAB(count10,2)==FIRSTLSYMBOLS(1,1))
     istatet=count10;
     end;
    count10=count10+1;
    end;
   INITIALSTATES(1,1:2)=STATETTAB(istatet,:);
   % Inserting initial states into STATEPROGTAB.
   VSTATEPROGTAB10=INITIALSTATES(1,1);
   VSTATEPROGTAB(1,1)=INITIALSTATES(1,2);
  else
   % If it's a partial response scheme, we consider all l symbols.
   while count10<=(m^l)
   % Just considering the states with 0 initial phases.
                    if  all(SRCHTAB(count10,2:l)==FIRSTLSYMBOLS(1,1:l-1))  &
all(SRCHTAB(count10,3:l+1)==FIRSTLSYMBOLS(1,2:l))
     istatet=count10;
     end;
    count10=count10+1;
    end;
   INITIALSTATES(1,1:2)=STATETTAB(istatet,:);
   % Inserting initial states into STATEPROGTAB.
   VSTATEPROGTAB(1,l-1:l)=INITIALSTATES;
  end; % if l==1
```

```
  % Inserting the second state into SPTAB and SPCCTAB.
  % The second state will be used as the initial state by the Viterbi decoder.
  SPTAB(1,1)=INITIALSTATES(1,2);
  SPCCTAB(1,1)=CCTAB(istatet,l);
  SPSYMTAB(1,1)=SRCHTAB(istatet,l+1);

  ACTUALPATHCCVAL(1,l)=CCTAB(istatet,l);

 elseif vcnt>=(l+1)
  % Growing the surviving paths.
  [SPTABpp,SPTABqq]=size(SPTAB);
  for count2=1:SPTABpp
   % For each final state of the surviving paths.

   if vcnt<=(l+nt-1)
    % If still building the initial survival paths.
    viterbipos=vcnt-l+1;
   else
      % If the initial survival paths have been grown, consider the last column of the
Viterbi matrix table.
    viterbipos=nt+1;
   end; % if, else

   % n is the state to be extended.
   n=SPTAB(count2,viterbipos-1);

   temp1=(n-1)*m+1;
   temp2=n*m;
   % Going through STATETTAB to extend SPTAB.
   for count3=temp1:temp2
    ESPTAB=[ESPTAB;[SPTAB(count2,:), STATETTAB(count3,2)]];
    ESPCCTAB=[ESPCCTAB; [SPCCTAB(count2,:), CCTAB(count3,vcnt)]];
                                                    ESPSYMTAB=[ESPSYMTAB;
[SPSYMTAB(count2,:),SRCHTAB(count3,l+1)]];
    end; % for count3=temp1:temp2
   end; % count2=1:SPTABpp

  [ESPTABpp, ESPTABqq]=size(ESPTAB);
  % SUMESPCCTAB should not be influenced by the first column since we
  % only consider the sum of nt column values.
  ESPCCTAB(:,1)=zeros(ESPTABpp,1);

  % Find the new surviving paths from the extended surviving paths.
  % Determine the cummulative cross correlation data for all extended paths.
  SUMESPCCTAB=sum(ESPCCTAB')';

  % For debugging purposes:
  %ESPTAB
```

```
%ESPCCTAB
%SUMESPCCTAB

% Initialising variables for no most likely path. This is really for vcnt>=(l+nt).
lpmax=mincumccval;
lpmaxpos=0;
spcount=0;

for count4=1:STATETABpp
 % For each state.
 % Initialising variables for no surviving paths.
 pmax=mincumccval;
 pmaxpos=0;
 for count5=1:ESPTABpp
  % For each extended surviving path.
   % We are looking for extended surviving paths with the same state we are now
considering.
                    if  STATETAB(count4,1)==ESPTAB(count5,viterbipos)  &
SUMESPCCTAB(count5,1)>pmax
   % If the state is the one we are considering and it is has the largest cum. CC value
so far.
    pmaxpos=count5;
    pmax=SUMESPCCTAB(count5,1);
            end;  % if STATETAB(count4,1)==ESPTAB(count5,viterbipos)  &
SUMESPCCTAB(count5,1)>pmax
  end; %  for count5=1:ESPTABpp
 if (pmaxpos~=0)
  % There's a surviving path for this state.

  % Keeping a count of surviving paths. This is really for vcnt>=(l+nt).
  spcount=spcount+1;

  TEMPSPTAB=[TEMPSPTAB; ESPTAB(pmaxpos,:)];
  TEMPSPCCTAB=[TEMPSPCCTAB; ESPCCTAB(pmaxpos,:)];
  TEMPSPSYMTAB=[TEMPSPSYMTAB; ESPSYMTAB(pmaxpos,:)];

  if pmax>lpmax
       % This surviving path is good enough to be ML path. This is really for
vcnt>=(l+nt).
    lpmaxpos=spcount;
    lpmax=pmax;
    end; % pmax>lpmax
  end; % (pmaxpos~=0)
  %lpmaxpos
  %lpmax
 end; %  for count4=1:STATETABpp
SPTAB=TEMPSPTAB;
SPCCTAB=TEMPSPCCTAB;
```

```
  SPSYMTAB=TEMPSPSYMTAB;

                                        ACTUALPATHCCVAL=[ACTUALPATHCCVAL,
SPCCTAB(lpmaxpos,viterbipos)];

  if vcnt>=(l+nt)
   % Storing away the states and output the detected symbols.
   MLPATH=SPTAB(lpmaxpos,:);
   MLSYM=SPSYMTAB(lpmaxpos,:);
   % Storing the oldest state on the most likely path into VSTATEPROGTAB.
   VSTATEPROGTAB(1,vcnt-nt)=MLPATH(1,2);
   % The current output symbol is the oldest symbol on the MLSYM.
   vcurrentsymbol=MLSYM(1,1);
   % Trimming the SPTAB, SPCCTAB AND SPSYMTAB.
   SPTAB=SPTAB(:,2:(nt+1));
   SPCCTAB=SPCCTAB(:,2:(nt+1));
   SPSYMTAB=SPSYMTAB(:,2:(nt+1));
   end; % if vcnt>=(l+nt)

  end; % if, elseif.

 end; % if (x==1)
y=vcurrentsymbol;


% PSEUDOCODE
% ==========
% Determine initial states
% ----------------------
% The INITIALSTATES (first 2 states) are determined from the first state transition.
% Skip the inputs until the end of the lth symbol interval.
% Look at the stored transmitted symbol sequence to determine which symbols were
transmitted.
% For more realism, the initial states should be worked out using ML paths.
% This would mean all possible paths (nt long) need to be considered.
% This would mean a slow detection of the initial states.
% Simply looking at the largest first column data of CCTAB is unreliable because small
noises
% can easily cause incorrect detection.
% For full response schemes, we consider only the first symbol. For partial response, the
first l symbols.
% We then store away the second initial state into SPTAB; it's corresponding CC data
into SPCCTAB;
% and the corresponding symbol in SPSYMTAB.
% For full response schemes, the first initial state is stored away in
VSTATEPROGTAB10 and the second
% in VSTATEPROGTAB(1,1).
% For full response schemes, the first two initial states are stored away
```

```
% in VSTATEPROGTAB(1,1:2).
% Note that for l==1, VSTATEPROGTAB cannot provide the symbols received because for any state
% transition, there will be more than one corresponding symbol.

% Extending the survival paths
% -------------------------
% We grow the surviving paths for each of the latest states in the SPTAB.
% We extend the paths by adding on all possible states reachable.
% To do this we look at STATETTAB.
% As we extend the SPTAB, we also extend SPCCTAB and SPSYMTAB.
% The extended tables are separate, namely ESPTAB, ESPCCTAB, ESPSYMTAB.
% For each state, we look for its the survival path by looking for the largest cummulative cross
% correlation value of the extended paths which have the same final state.
% While we look for the survival paths, we should also record the survival paths with the largest
% cummulative cross correlation value for the most likely path.
% Define the new survival paths tables, SPTAB, SPCCTAB & SPSYMTAB.

% Output the detected symbols
% -------------------------
% If we have grown the initial survival paths (ie.vcnt>=(nt+1)), we output the detected symbols.
% The current detected symbol is the oldest symbol of the most likely path.
% The oldest state of the most likely path is stored away in VSTATEPROGTAB.
% SPTAB, SPCCTAB & SPSYMTAB are trimmed by removing the oldest column.
```

Test Functions

chktrav.m

```
% FILE: chktrav.m
% ==============
% From the symbol sequence, this program generates:
% 1. the states traversed
% 2. the STATETTAB position traversed
% 3. the ROM's phase waveform
% This program is to be executed after simulation only.

% Note:
% currentpos holds the position of the currentphase - not the lth symbol.
% STATETTABPOSTRAV is processed one step in advance to STATESTRAV -
% be aware when looking for values at currentpos instance.
```

```
% Input variables.
global SYMBOLS;
global sd;
global l;
global STATETAB;
global SRCHTAB;
global h;

% Output variables.
% Array of symbols.
global SYMS;
% States traversed.
global STATESTRAV;
% State transition table position traversed.
global STATETTABPOSTRAV;
% ROM's phase waveform.
global ROMPHASEWAVEFORM;
global ROM;

STATESTRAV=[];
STATETTABPOSTRAV=[];
ROMPHASEWAVEFORM=[];

% Local variables.
POS=[];
cumphase=0;
currentpos=0;
error=0.001;
LSYMS=[];
nextsym=0;

% Obtaining the array of symbols.
TEMP=[1:round(size(SYMBOLS,2)/sd)-1];
POS=sd*TEMP;
SYMS=SYMBOLS(POS);
sizeSYMS=size(SYMS,2);

currentphase=cumphase;

if l>1
  while (currentpos+l<=sizeSYMS)
    % Bounding the phase within 2*pi.
    currentphase=rem(currentphase,2*pi);
    % Positiving any negative values.
    currentphase=rem(currentphase+2*pi,2*pi);
    % Zeroing any values close to 0 and 2*pi.
    if (abs(currentphase-0)<error) | (abs(currentphase-2*pi)<error)
      currentphase=0;
```

```
    end;

   % Getting next l symbols.
   LSYMS=SYMS(1,currentpos+1:l+currentpos);
   % Getting next l-1 symbols.
   LMIN1SYMS=LSYMS(1,1:l-1);

   % Searching STATETAB to find the state.
   sizeSTATETABpp=size(STATETAB,1);
   STATETABpos=1;
              while   (~all(LMIN1SYMS==STATETAB(STATETABpos,3:2+l-1))   |
(abs(currentphase-STATETAB(STATETABpos,2))>error))                          &
(STATETABpos<sizeSTATETABpp)
    STATETABpos=STATETABpos+1;
    end;

   % Searching SRCHTAB for the position of the corresponding ROM.
   sizeSRCHTABpp=size(SRCHTAB,1);
   SRCHTABpos=1;
       while  (~all(LSYMS==SRCHTAB(SRCHTABpos,2:1+l))  |  (abs(currentphase-
SRCHTAB(SRCHTABpos,1))>error)) & (SRCHTABpos<sizeSRCHTABpp)
    SRCHTABpos=SRCHTABpos+1;
    end;

   STATESTRAV=[STATESTRAV,STATETABpos];
   STATETTABPOSTRAV=[STATETTABPOSTRAV,SRCHTABpos];

   % Determine the ROM's phase waveform.
                          ROMPHASEWAVEFORM=[ROMPHASEWAVEFORM,
cumphase+(ROM(SRCHTABpos, 2:1+sd)-SRCHTAB(SRCHTABpos,1))];

   currentphase=currentphase+pi*h*LSYMS(1,1);
   cumphase=cumphase+pi*h*LSYMS(1,1);
   currentpos=currentpos+1;

   end;

   % The first state is determined at the (l-1)th symbol interval.
   STATESTRAV=[zeros(1,l-2),STATESTRAV];
   % The first state transition is determined at the lth symbol interval.
   STATETTABPOSTRAV=[zeros(1,l-1),STATETTABPOSTRAV];

   % The first ROM output is at the lth symbol interval.
   ROMPHASEWAVEFORM=[zeros(1,(l-1)*sd),ROMPHASEWAVEFORM];


elseif (l==1)
  while (currentpos<=sizeSYMS-1)
```

```
  % Bounding the phase within 2*pi.
  currentphase=rem(cumphase,2*pi);
  % Positiving any negative values.
  currentphase=rem(currentphase+2*pi,2*pi);
  % Getting next symbol.
  nextsym=SYMS(1,currentpos+1);

  % Searching STATETAB to find the state.
  sizeSTATETABpp=size(STATETAB,1);
  STATETABpos=1;
            while (abs(currentphase-STATETAB(STATETABpos,2))>error) &
(STATETABpos<sizeSTATETABpp)
   STATETABpos=STATETABpos+1;
   end;

  % Searching SRCHTAB for the position of the corresponding ROM.
  sizeSRCHTABpp=size(SRCHTAB,1);
  SRCHTABpos=1;
          while (nextsym~=SRCHTAB(SRCHTABpos,2) | (abs(currentphase-
SRCHTAB(SRCHTABpos,1))>error)) & (SRCHTABpos<sizeSRCHTABpp)
   SRCHTABpos=SRCHTABpos+1;
   end;

  STATESTRAV=[STATESTRAV,STATETABpos];
  STATETTABPOSTRAV=[STATETTABPOSTRAV,SRCHTABpos];

  % Determine the ROM's phase waveform.
                          ROMPHASEWAVEFORM=[ROMPHASEWAVEFORM,
cumphase+(ROM(SRCHTABpos, 2:1+sd)-SRCHTAB(SRCHTABpos,1))];

  currentphase=currentphase+pi*h*nextsym;
  cumphase=cumphase+pi*h*nextsym;
  currentpos=currentpos+1;
  end;

  % The first state is determined at the 0th symbol interval.
  sizeSTATESTRAV=size(STATESTRAV,2);
  STATESTRAV=STATESTRAV(1,2:sizeSTATESTRAV);
  % The first state transition is determined at the 1st symbol interval.
  % The first ROM output is at the 1st symbol interval.

end;

close all;
sizeROMPHASEWAVEFORM=size(ROMPHASEWAVEFORM,2);
T=[1:sizeROMPHASEWAVEFORM];
plot(T,ROMPHASEWAVEFORM(1,1:sizeROMPHASEWAVEFORM),'r',T,PHASEW
AVEFORM(1,1:sizeROMPHASEWAVEFORM),'b');
```

```
corrcoef(ROMPHASEWAVEFORM(1,(l-
1)*sd+1:sizeROMPHASEWAVEFORM),PHASEWAVEFORM(1,(l-
1)*sd+1:sizeROMPHASEWAVEFORM))
```

compcomp.m

```
% FILE: compcomp.m
% ================
% Compares the received inphase and quadrature components with that
% of the transmitted signal.
% This program is to be executed after simulation only.

close all;
sizeRINPHASE=size(RINPHASE,2);
RIP=RINPHASE(1,correldelay+1:sizeRINPHASE-1);
RQ=RQUAD(1,correldelay+1:sizeRINPHASE-1);

COSP=cos(PHASEWAVEFORM(1,1:sizeRINPHASE-correldelay-1));
SINP=sin(PHASEWAVEFORM(1,1:sizeRINPHASE-correldelay-1));

T=[1:sizeRINPHASE-correldelay-1],
plot(T,COSP,'b',T,RIP,'r');
figure
plot(T,SINP,'b'.T,RQ,'r');
```

compcorr.m

```
% FILE: compcorr.m
% ================
% Used for checking the cross correlator.
% For comparing the actual cross correlation with the
% cross correlator's output for a particular symbol interval.

% Input variables.
global correldelay;
global sd;

% User parameters.
% The position on the CCTAB where the output needs to be checked.
syminterval=7;
rompos=2;

T=[1:sd];
```

```
COSROM=cos(ROM(rompos,2:sd+1));
RINP=RINPHASE(1,correldelay+sd*(syminterval-1)+1:correldelay+syminterval*sd);
SINROM=sin(ROM(rompos,2:sd+1));
RQ=RQUAD(1,correldelay+sd*(syminterval-1)+1:correldelay+syminterval*sd);

A=corrcoef(COSROM+0.000001,RINP+0.000001);
B=corrcoef(SINROM+0.000001,RQ+0.000001);
C=(A+B)/2
CCTAB(rompos,syminterval)

close all
plot(T,COSROM, 'r', T,RINP, 'b');
figure
plot(T,SINROM, 'r', T,RQ, 'b');
```

compsym.m

```
% FILE: compsym.m
% ==============
% Compares the original symbols with the received symbols
% by plotting both.

close all;
plot(SYMBOLS)
figure
plot(RSYMBOLS)
```